THE UNIVERSITY
of EDINBURGH

Ray-traced Radiative Transfer on Massively Threaded Architectures

Samuel Thomson

Doctor of Philosophy

The University of Edinburgh

2018

Abstract

In this thesis, I apply techniques from the field of computer graphics to ray tracing in astrophysical simulations, and introduce the GRACE software library. This is combined with an extant radiative transfer solver to produce a new package, TARANIS. It allows for fully-parallel particle updates via per-particle accumulation of rates, followed by a forward Euler integration step, and is manifestly photon-conserving. To my knowledge, TARANIS is the first ray-traced radiative transfer code to run on graphics processing units and target cosmological-scale smooth particle hydrodynamics (SPH) datasets.

A significant optimization effort is undertaken in developing GRACE. Contrary to typical results in computer graphics, it is found that the bounding volume hierarchies (BVHs) used to accelerate the ray tracing procedure need not be of high quality; as a result, extremely fast BVH construction times are possible (< 0.02 microseconds per particle in an SPH dataset). I show that this exceeds the performance researchers might expect from CPU codes by at least an order of magnitude, and compares favourably to a state-of-the-art ray tracing solution. Similar results are found for the ray-tracing itself, where again techniques from computer graphics are examined for effectiveness with SPH datasets, and new optimizations proposed. For high per-source ray counts ($\gtrsim 10^4$), GRACE can reduce ray tracing run times by up to two orders of magnitude compared to extant CPU solutions developed within the astrophysics community, and by a factor of a few compared to a state-of-the-art solution.

TARANIS is shown to produce expected results in a suite of *de facto* cosmological radiative transfer tests cases. For some cases, it currently out-performs a serial, CPU-based alternative by a factor of a few. Unfortunately, for the most realistic test its performance is extremely poor, making the current TARANIS code unsuitable for cosmological radiative transfer. The primary reason for this failing is found to be a small minority of particles which always dominate the timestep criteria. Several plausible routes to mitigate this problem, while retaining parallelism, are put forward.

Lay summary

Reionization is a significant event in the universe's 13-billion-year history, beginning when it was 500 million years old and ending 500 million years later. It marks the point at which the first sources of light began to emerge, in the form of stars and galaxies. The electromagnetic radiation, or photons, emitted by these early counterparts to our Sun and Milky Way galaxy caused a fundamental change in the state of the surrounding (mostly-hydrogen) gas. Understanding exactly how this process unfolded is therefore critical to our understanding of how the universe came to be as it appears today.

It is only over the last decade, with tools like the Hubble Space Telescope, that we have been able to observe these early galaxies. And, even now, observations are limited to the tail-end of reionization. Outside of pure theory, much work has therefore focussed on simulating the process of reionization on supercomputers. This too has proved extremely challenging, and requires significant computational resources to achieve with sufficient accuracy and scale.

To that end, in this thesis I present two pieces of software which are intended to reduce the time and resources required by these simulations.

The first is GRACE, a *ray tracing* code. Ray tracing is a particularly accurate way of modelling the path that photons, or light, follow. It is often used to produce incredibly realistic, computer-generated images. In the same vein, it is also an excellent way to model photons emitted in the early universe. Unfortunately this accuracy comes at a high cost: ray tracing takes a long time for a computer to perform. Since its introduction in the late 1960s, significant effort has been undertaken in the field of computer graphics (CG) to improve algorithms for ray tracing. Despite its applicability, this body of work has thus far received limited attention in astrophysics. In developing GRACE, I have utilized many of the techniques developed for CG, in addition to those of my own, for ray tracing in astrophysical simulations. This is extremely successful, with GRACE being up to 100 times faster than current solutions in astrophysics for the same task; it also

compares very well to state-of-the-art solutions, typically being a few times faster when applied to astrophysical simulations.

The second piece of software is TARANIS. Given the photon paths produced by GRACE, it solves the equations describing the state of the (mostly-hydrogen) gas of the early universe. TARANIS is moderately successful in its stated goal of reducing the resources required, but still leaves much room for improvement. I therefore conclude with several promising suggestions for future work.

Declaration

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where stated otherwise by reference or acknowledgement, the work presented is entirely my own.

Samuel Thomson, January 2018

Acknowledgements

First and foremost, I would like to thank Eric, without whose unyielding air of calm I would surely have cracked. Secondly, and frankly also foremost, I would like to thank Paula for all the things she knows full well I had no idea she'd done, and which she continues to do.

Finally, thank you to my parents, to Hari, and to my office comrades Ali and Marco for making sure I was never alone.

This work was supported by the STFC.

Contents

Abstract i

Lay summary iii

Declaration v

Acknowledgements vii

Contents xiii

List of Algorithms xv

List of Listings xvii

List of Figures xxi

List of Tables xxiv

1 Introduction 1

2 Computational Radiative Transfer in Astrophysics 3

1 Reionization 3

1.1 Observations 4

1.2 Numerical Results 6

2 Radiative Transfer Theory 9

2.1 The Fundamentals 9

2.2 The On-the-Spot Approximation 12

2.3 The 3D Equation of Radiative Transfer 13
2.4 The Ionization Equations 15
2.5 The Temperature Equation 16
3 Numerical Simulations in Astrophysics 17
3.1 N-body Dynamics 17
3.2 Adaptive Mesh Refinement 19
3.3 Smoothed Particle Hydrodynamics 21
3.4 Moving mesh codes 23
4 Numerical Radiative Transfer 24
4.1 Moment Methods 24
4.2 Direct Photon Transfer 27
SimpleX 27
TRAPHIC 29
4.3 Ray Traced Radiative Transfer 29
SPHRAY 32
P-SPHRAY 34

3 Ray Tracing 39
1 Introduction 39
2 Ray Tracing Fundamentals 40
2.1 Ray and axis-aligned box intersection 40
2.2 Ray-sphere intersection 43
3 Ray Tracing Techniques from Computer Graphics 44
3.1 Acceleration Structures 45
3.2 The Surface Area Heuristic 49
3.3 The degree of acceleration structures 50
3.4 Ray Tracing in Astrophysics 51
4 Graphics Processing Units 54
4.1 CUDA and NVIDIA Hardware 54
4.2 On-GPU Bounding Volume Hierarchy Construction 58
LBVH 59
The Morton Key 61
5 Summary 63

4 GRACE: A GPU-accelerated ray tracing code 65

1 Introduction 65

2 Considerations for GPU Code 66

3 Considerations for Ray Tracing SPH Datasets 69

4 Parallel Primitives 73

5 The Acceleration Structure: ALBVH 76

5.1 ALBVH Algorithm 77

5.2 Initial ALBVH Implementation 80

5.3 Multiple Primitives per Leaf 84

6 Optimizing ALBVH Construction 91

6.1 Optimized Wide-Leaf Implementation 92

6.2 Optimized Node Building 101

7 Ray Tracing Implementation 111

7.1 Casting rays 111

7.2 Computing the SPH integral 117

7.3 BVH traversal 120

Naïve traversal 120

Memory layout of nodes 122

Packet traversal 122

Postponed traversal 125

Stackless traversal 126

Improving coalescing on memory stores 128

Ray-AABB intersection tests 130

Miscellaneous traversal optimizations 133

8 Verification 135

9 Performance 137

9.1 CPU Implementations 138

9.2 Test Datasets 139

9.3 Tree Build Performance 139

9.4 Ray Tracing Performance 141

9.5 The effectiveness of packet traversal and limits to performance 144

10 Generalizing GRACE to other datasets 153
10.1 Application to triangle meshes 155
10.2 Optimization of closest-intersection traversal 156
11 Conclusion 162

5 TARANIS: A GPU-accelerated numerical radiative transfer code 165
1 Introduction 165
2 Contributions 165
3 A parallel radiative transfer algorithm 166
4 Radiative transfer implementation 168
4.1 Ionization state 168
4.2 Optical depth 169
4.3 Photoionization rates 171
4.4 Photoheating and cooling 175
4.5 Euler integration timestep 179
5 The cosmological radiative transfer comparison project 179
5.1 Test I: pure hydrogen isothermal HII region expansion 181
Initial conditions 181
Results and discussion 182
5.2 Test II: HII region expansion: the temperature state 185
Results and discussion 185
5.3 Test III: Ionization front trapping in a dense clump and the formation of a shadow 193
Initial conditions 193
Results and discussion 195
5.4 Test IV 202
Initial conditions 203
Results and discussion 203
5.5 Convergence 210
Expected ray count requirement 210
Test II 211
Test III 216

Test IV 218
5.6 Performance 223
Test II 224
Test III 226
Test IV 228
5.7 Routes to improved performance 230
6 Conclusion 234

Appendices

A Hydrogen Ionizing Photon Flux of a Black Body 239

B Generating Uniform Random Directions 243
1 Uniform Directions Within a Solid Angle 243
2 Uniform Directions On the Unit Sphere 249

C Block-wide exclusive scan sum of booleans 251

D Equivalence and noninferiority tests 253
1 Equivalence test for normal data 253
2 Noninferiority test for non-normal data 254

E Discrepancies with Iliev et al. 2006 257

List of Algorithms

4.1 Wide leaf detection 86
4.2 Queue-filling climb 104

List of Listings

4.1 Structure-of-arrays 68
4.2 Array-of-structures 68
4.3 A reduction 73
4.4 An inclusive scan 73
4.5 An exclusive scan. 73
4.6 Node to parent climb 83
4.7 Node to parent climb without fence 95
4.8 Stack-based traversal 121
4.9 Coalesced writes 129
C.1 Warp-wide exclusive scan sum 251
C.2 Block-wide exclusive scan sum 252

List of Figures

2.1 P-SPHRAY ionization state radial profiles 35
2.2 P-SPHRAY ionization and temperature state radial profiles 37

3.1 Ray-box intersection 41
3.2 Ray-sphere intersection 43
3.3 Long and short characteristics ray tracing 52
3.4 Optimal long characteristics ray tracing 52
3.5 Morton and Hilbert curves 60
3.6 Computing a Morton key in 3D 61
3.7 IEEE 754 binary32 62

4.1 Conditions for partial ray-sphere intersections 70
4.2 Intersection data layout 74
4.3 Segmented scan 74
4.4 Segmented sort 75
4.5 ALBVH hierarchy 79
4.6 Climbing a tree hierarchy 82
4.7 Wide leaf conditions 84
4.8 Wide-leaf building 93
4.9 Non-hierarchical leaf building 97
4.10 Optimized leaf-build performance 98
4.11 Iterative node building 103
4.12 Optimized node-build performance for $\phi_{\max} = 32$ 109
4.13 Optimized node-build performance for $\phi_{\max} = 1$ 110
4.14 Ray K-statistics 116
4.15 Ray-sphere SPH integral 118
4.16 SPH kernel line integral approximations 119

4.17 Leaf size traversal performance 124
4.18 Depth-first traversal order 127
4.19 Cumulative traversal performance 142
4.20 All-intersections traversal performance 145
4.21 N_{rays} per source traversal performance 148
4.22 Number of leaf requests per warp 152
4.23 CG scene renders 156

5.1 CRTCP Tests results legend 181
5.2 CRTCP Test 1 x_{HI} and Strömgren radius 182
5.3 CRTCP Test 1 x_{H} radial profiles 184
5.4 CRTCP Test 1 x_{HI} histograms 184
5.5 CRTCP Test 2 x_{HI} and ionization front radius 186
5.6 CRTCP Test 2 T slices at 10 Myr 188
5.7 CRTCP Test 2 x_{H} radial profiles 189
5.8 CRTCP Test 2 T radial profiles 190
5.9 CRTCP Test 2 x_{H} and T radial profiles 191
5.10 CRTCP Test 2 x_{HII} histograms 192
5.11 CRTCP Test 2 T histograms 192
5.12 CRTCP Test 3 x_{H} and T radial profiles 197
5.13 CRTCP Test 3 x_{H} and T radial profiles 198
5.14 CRTCP Test 3 x_{H} slices at 15 Myr 198
5.15 CRTCP Test 3 T slices at 15 Myr 199
5.16 CRTCP Test 3 x_{HII} histograms at 15 Myr 200
5.17 CRTCP Test 3 T histograms at 1 Myr 201
5.18 CRTCP Test 3 T histograms at 15 Myr 201
5.19 CRTCP Test 4 x_{HI} slices at 0.05 Myr 204
5.20 CRTCP Test 4 T slices at 0.05 Myr 205
5.21 CRTCP Test 4 x_{HI} slices at 0.4 Myr 206
5.22 CRTCP Test 4 T slices at 0.4 Myr 207
5.23 CRTCP Test 4 x_{HII} fractions over time 208
5.24 CRTCP Test 4 x_{HI} histograms 209
5.25 CRTCP Test 4 T histograms 209
5.26 N_{rays} per timestep Test 2 results legend 211

5.27 CRTCP Test 2 neutral-fraction slices at 10 Myr convergence 212
5.28 CRTCP Test 2 temperature slices at 10 Myr convergence 212
5.29 CRTCP Test 2 neutral-fraction slices at 500 Myr convergence 213
5.30 CRTCP Test 2 temperature slices at 500 Myr convergence 213
5.31 CRTCP Test 2 T radial profiles convergence 214
5.32 CRTCP Test 2 T histograms for various N_{rays} 215
5.33 N_{rays} and timestep factor Test 3 results legend 216
5.34 CRTCP Test 3 neutral-fraction slices at 5 Myr convergence 217
5.35 CRTCP Test 3 temperature slices at 5 Myr convergence 217
5.36 CRTCP Test 3 x_{HI} and T radial profiles convergence 218
5.37 CRTCP Test 3 x_{HII} histograms convergence at $t = 5$ Myr 219
5.38 CRTCP Test 3 T histograms convergence at $t = 5$ Myr 219
5.40 CRTCP Test 4 neutral-fraction slices at 0.1 Myr convergence 221
5.41 CRTCP Test 4 temperature slices at 0.1 Myr convergence 221
5.39 CRTCP Test 4 performance results legend 221
5.42 CRTCP Test 4 neutral-fraction slices at 0.4 Myr convergence 222
5.43 CRTCP Test 4 temperature slices at 0.4 Myr convergence 222
5.44 CRTCP Test 4 x_{HI} histograms convergence 222
5.45 CRTCP Test 4 T histograms convergence 223
5.46 TARANIS Test 2 performance 224
5.47 TARANIS Test 2 timestep size 226
5.48 TARANIS Test 3 performance 227
5.49 TARANIS Test 3 timestep size 228
5.50 TARANIS Test 3 performance 229
5.51 TARANIS Test 4 timestep size 229
5.52 CRTCP Test 2 x_{HI} vs timescale 231
5.53 CRTCP Test 2 timescale histograms 232

List of Tables

4.1 Thrust and SGPU segmented operation performance 77
4.2 GPU hardware comparison 88
4.3 LBVH implementation performance 89
4.4 LBVH implementation performance sans sort 90
4.5 ALBVH-build timing data 91
4.6 Summary of leaf-build optimizations 99
4.7 Summary of node-build implementations 108
4.8 Optimized LBVH implementation performance 111
4.9 Time to generate random numbers 114
4.10 Ray noninferior to uniform sample results 117
4.11 Coalesced writes performance 130
4.12 Ray-AABB intersection test performance 132
4.13 SPH volume integrals 137
4.14 Test SPH datasets 139
4.15 Tree build performance 141
4.16 Closest-hit packet traversal 158
4.17 Closest-hit non-packet traversal 159
4.18 Closest-hit ALK kernel performance 160
4.19 Closest-hit non-packet SPH traversal 161

5.1 Recombination coefficients 169
5.2 Absorption cross section fits 171
5.3 Gaussian quadrature abscissas and weights 176
5.4 Recombination cooling coefficients 178
5.5 Collisional excitation cooling fits 178
5.6 Test II N_{rays} for correct sampling 211

5.7 TARANIS Test 2 runtime contributions 225

1

Introduction

Radiative transfer concerns itself with the propagation of electromagnetic radiation (i.e. photons) through an interacting medium. Generally, absorption, emission and scattering processes all play a role. Within the context of astrophysics there are several regimes, in which each of these processes has a greater or lesser effect on the physics.

For very high opacity situations scattering and absorption are particularly important; this is the case when modelling radiation transport within galaxies and supernovae. Cosmic dust grains are efficient in scattering and absorbing both UV and optical photons, and dust radiative transfer is an entire topic unto itself, one which is not covered here (though see e.g. Steinacker et al. (2013) for a recent review).

The focus of interest here lies in larger scales, where the opacity is lower and scattering becomes negligible. Additionally, the emission tends to be dominated by bright point sources, such as stars, galaxies and quasars. The Epoch of Reionization (EoR) marks the point in the Universe's history at which it transitioned from an almost entirely neutral to almost entirely ionized state. The EoR is, ultimately, the target of studies into radiative transfer on cosmological scales.

On the observational front, significant progress has recently been made in determining exactly which objects were responsible for reionization. Robertson et al. (2010) present a clear and succinct review of results obtained from observations of the earliest visible galaxies. Present evidence seems to suggest that more abundant, smaller, lower mass galaxies drove reionization, but this is by no means clear-cut. Forthcoming 21 cm observations (a spectral line due to a hyperfine transition of neutral hydrogen) promise

to greatly improve on our current observational view of reionization (Furlanetto et al., 2009). However, the Square Kilometer Array (SKA), the largest of these ventures, is still several years away.

In terms of numerical simulations, the (cosmological) equation of 3D radiative transfer is somewhat intractable, being seven-dimensional (three spatial co-ordinates, two angles, frequency and time). Thankfully, it is well-approximated by solving the 1D solution along many *rays*, which represent photons (or photon packets), emitted by sources of ionizing radiation. This technique is known as ray tracing, and is in principle the most accurate method of simulating radiative transfer. Unfortunately, it has a high computational cost, and in particular scales with the number of sources — this is not ideal for the case of reionization, with upwards of 10^4 sources (Paardekooper et al., 2012) in a simulation volume. Despite the high cost, ray tracing still presents a very parallel and independent problem. To this end, I have developed the ray tracing library GRACE, which uses graphics processing units (GPUs — powerful compute units with as many as $\sim 3,000$ simple cores, capable of performing the same operation on different data simultaneously) to accelerate the ray tracing process.

This thesis is layed out as follows; in Chapter 2 numerical radiative transfer is introduced, with a focus on its application in smoothed particle hydrodynamics (SPH); in Chapter 3 ray tracing is covered, from the perspective of both computer graphics and computational astrophysics; in Chapter 4 the GRACE library is presented, including key algorithms and optimizations, and its generalization to non-SPH datasets; finally, in Chapter 5 TARANIS is introduced, the union of GRACE and an extant numerical radiative transfer code which also runs on the graphics processing unit (GPU), and its results are verified via comparison to other codes on a set of standard numerical tests.

2

Computational Radiative Transfer in Astrophysics

In this chapter, key concepts from the field of numerical radiative transfer are introduced, from the perspective of astrophysics. This provides both background for Chapter 5 and context for the more computationally-focussed Chapters 3 and 4.

Section 1 frames radiative transfer simulations within the wider picture; Section 2 covers the underlying theory and presents the relevant equations; Section 3 summarizes the current techniques in computational astrophysics within which a numerical radiative transfer scheme must operate; and, finally, Section 4 discusses the most common approaches to implementing radiative transfer, as well as describing and evaluating some of the extant solutions to the problem.

1 Reionization

The Epoch of Reionization (EoR) represents the last major phase-change of the universe. Prior to *reionization*, structure formation and evolution was dominated by dark matter; post-reionization, the role of gas-physics in structure evolution became substantially more important, particularly on small scales.

Shortly after the Big Bang, around redshift $z_{\text{rec}} \approx 1,100$ ($t_{\text{rec}} \approx 0.4$ Myr), the temperature of the universe fell to a sufficiently cool temperature for electrons and protons to (re-)combine, forming neutral hydrogen and helium (and a negligible amount

of some heavier elements). At this point the baryons decoupled from the photons, leaving a surface of last scattering, now visible as the cosmic microwave background (CMB); this decoupling also made the universe *optically thin* to photons below the ionization threshold, and began the so-called *dark ages* (Zaroubi, 2013). The dark ages were brought to an end with the appearance of the first sources of light. This marks the beginning of the EoR (which commonly refers only to *hydrogen* reionization). During the EoR, the universe was a patchwork of HII regions, centred around the first objects, in an otherwise neutral-hydrogen medium. As more ionizing sources formed, the HII regions grew, overlapped, and eventually expanded to fill the entire universe. The ionization process also results in a negative feedback loop for source formation: ionization leads to heating, which provides gas-pressure support against collapse, and it reduces the cooling efficiency of the gas, which therefore also cools more slowly as it collapses.

While the above picture may seem relatively straightforward, there are still many unknowns. For example, it is not certain which objects played the dominant role; further, it is not known which areas were ionized first — low density (where there are fewer ionizing sources, but also a lower rate of *recombinations*), or high density (where there are more ionizing sources, but a higher rate of recombinations). The former is referred to as inside-out reionization, the latter outside-in.

1.1 Observations

Both neutral and ionized hydrogen are useful observational targets for constraining reionization. The Thomson optical depth is a probe for HII, determined from the scattering effect of electrons on CMB photons (Planck Collaboration, 2014). For observation of neutral hydrogen we have two options: absorption of Lyman-alpha photons (Lyα, the $n = 2 \rightarrow 1$ hydrogen transition) and the 21 cm line, a hyperfine transition of neutral hydrogen (between parallel and anti-parallel electron-proton spin states, the latter having slightly lower energy, so called because the energy difference corresponds to radio waves at a wavelength of 21 cm).

The timespan of the EoR is not yet precisely known; for hydrogen, it is thought to be approximately $z \sim 11 - 6$ (Paardekooper et al., 2012). The lower limit, marking the end of reionization, comes from the Gunn-Peterson trough. Essentially, whilst there still exists neutral hydrogen in the intergalactic medium (IGM), we expect it to absorb

radiation at wavelengths below the Lyα limit. This results in a trough in the spectra of distant quasars, appearing at approximately $z \sim 6$. At lower redshifts the trough is not apparent, suggesting that the universe is by then fully ionized. One upper limit, marking the beginning of reionization, comes from the aforementioned CMB data and suggests substantial ionization by $z \sim 11$ (Planck Collaboration, 2014).

There are several candidates for the objects which reionized the universe, but the current model is that the ionizing radiation was produced by the first stars and galaxies. As a result, a focus of recent work has been to ascertain whether or not this is actually feasible. A concise review of the topic is given by Robertson et al. (2010), the essentials of which are covered here.

To reionize the universe fully, at a miniumum, the integrated history of ultraviolet (UV) photons emitted by galaxies must contain *more* than one ionizing photon per hydrogen atom (to account for recombinations). Such *high-redshift* galaxies may be identified via a break in their spectra: even small amounts of neutral hydrogen can absorb all light at wavelengths shorter than the Lyα line; as a result, they are often referred to as *Lyman-break galaxies*. While the rest-frame wavelength of the Lyα line is in the UV (121.6 nm), detection of redshifted Lyα emissions from $z \sim 6$ sources requires use of an infrared detector, such as that included in the Wide Field Camera 3 (WFC3/IR) on the Hubble Space Telescope (HST). Recent results (Robertson et al., 2013) suggest that, in order for galaxies to reionize the universe by $z \sim 6$ *and* produce the observed Thomson optical depth, galaxy formation must occur as early as $z \sim 12 - 15$. However, the most recent Planck data (Planck Collaboration, 2016) contains a slightly reduced optical depth. Robertson et al. (2015) have been able to reconcile this reduced value with luminosity distributions derived from the latest HST imaging, resulting in a reionization epoch spanning $6 \lesssim z \lesssim 10$, reducing the prior requirements for a significant number of high redshift galaxies at $z > 10$. Their analysis supports a reionization driven primarily by star-forming galaxies.

The 21 cm background is an ideal probe of reionization, and we can in principle look to much higher redshifts (well into the cosmic dark ages). Further, we are able to image over the entire sky, as with the CMB, and as it is a spectral line, we also obtain redshift information. The nature of ionizing sources will have a strong effect on the 21 cm power spectrum (Furlanetto et al., 2009). For example, stars produce well-defined HII regions, while those due to quasars, with their power-law spectra, are much more

diffuse (Zaroubi and Silk, 2005), and massive galaxies should produce larger ionized regions — these different scenarios all leave distinct imprints in the 21 cm signal. The formation of the very first structures in the universe may be inaccessible even to James Webb Space Telescope (JWST), but again is in principle observable from 21 cm.

21 cm arrays (detectors) are a particularly recent development, with many radio interferometers (the 21 cm signal from the EoR is redshifted to wavelengths of metres) only just beginning observations (LOFAR[1], PAPER[2], MWA[3] and, in the future, SKA[4]). Parsons et al. (2014), with three months of observing on a 32-antennae deployment of PAPER, have 'suggestive evidence' that by $z = 7.7$ the neutral IGM has been warmed from its cold primordial state. We expect more extensive results from 21 cm surveys in the near future.

1.2 Numerical Results

Cosmological radiative transfer codes, such as those discussed in Section 4, are aimed at modelling the reionization problem. Due to its complexity, it is common to post-process the output of a gravity-hydrodynamics simulation. Recently, Paardekooper et al. (2012) did just this, using the radiative transfer code SimpleX (see Section 4.2) to estimate the contribution of stars and galaxies to the reionization process. They find that lower mass galaxies (with stellar masses $10^5 M_\odot < M_* < 10^6 M_\odot$) drive the initial ionization process, from $z \sim 15 - 10$. This ionization is maintained at later times ($z \lesssim 10$) by the more massive galaxies, as they start to appear more frequently. Additionally, Population III stars appear to be too short-lived and infrequent to make a significant contribution. Their conclusions are roughly in-line with the general picture of reionization we currently have from observations, but until JWST data is available we are unlikely to be able to observe the proto-galaxies potentially driving reionization.

Ciardi et al. (2012) examine the oft-neglected effect of helium on hydrogen reionization. The process is slightly delayed, as expected, since some ionizing photons are absorbed by helium rather than hydrogen; more importantly, they find a $\sim 20\%$ increase in the temperature of the IGM as compared to the hydrogen-only case. Comparison to observations of the IGM temperature at $z \sim 5 - 6$ suggest that reionization was

[1] http://www.lofar.org
[2] http://eor.berkeley.edu/
[3] http://www.mwatelescope.org/
[4] http://www.skatelescope.org/

driven primarily by Population II sources. Similarly to Paardekooper et al. (2012), they also find that faint, currently undetectable galaxies are required to produce the ionizing emissivity used in the simulation (which was itself consistent, by design, with observational measurements of the Thomson optical depth).

Feng et al. (2013) examine results from post-processing regions around high-redshift ($z \sim 8$) quasars using a modified version of the SPHRAY code, P-SPHRAY (see Section 4.3 for a brief discussion of the current state of this code). While they find that the ionization regions produced are not sufficiently large to cover the entire IGM, their relative isolation makes then potentially interesting candidates for future 21 cm observations.

Shapiro et al. (2012) have post-processed N-body hydrodynamics runs using the then-latest version of the C^2-RAY code (Mellema et al., 2006; Friedrich et al., 2012) in an effort to identify which galaxies were mainly responsible for reionization, and how observations relate to the different scenarios. Galaxies are grouped into three bins, defined by their dark matter halo mass: minihaloes (MH, $10^5 M_\odot < M < 10^8 M_\odot$), low-mass atomic-cooling haloes (LMACH, $10^8 M_\odot < M < 10^9_\odot$) and high-mass atomic-cooling haloes (HMACH, $M > 10^9 M_\odot$). (MHs have a virial temperature $< 10^4$ K; since atomic cooling in primordial hydrogen-helium gas is ineffective at these temperatures, they require molecular H_2 cooling to enable star formation.) It is found that 21 cm power spectrum fluctuations from e.g. MWA have the potential to distinguish the HMACH-only reionization case from the HMACH, LMACH (and optionally MH) case. CMB data can similarly distinguish the HMACH-only from the others, but no better — unless reionization ended as late as $z < 7$ (not unreasonable, based on observational evidence), in which case fluctuations in polarization CMB data caused by MHs should be detectable.

Hutter et al. (2017) find reionization to proceed inside-out, with the ionization fractions being highest in dense knots, followed by less dense filaments, sheets, and finally the low-density voids, once the IGM is $> 10\%$ neutral. They also predict strong anti-correlations between visible Lyman-α emitters (LAEs) (whose emissions are affected by the presence of even trace levels of neutral hydrogen) and 21 cm emissions of neutral hydrogen; further, they believe that forthcoming 21 cm observations will be sufficiently sensitive to lend support to this scenario, detecting a lower 21 cm brightness temperature in the densest regions (specifically, 1000 hours of low-frequency SKA Phase 1 data).

Bauer et al. (2015) find reionization to proceed inside-out, and also determine that ordinary stellar sources produce enough ionizing photons in total whilst maintaining

consistency with the Planck Collaboration (2016) results for the cumulative optical depth due to Thomson scattering, but only for models with the most optimistically-high escape fractions at high redshift (those which scale as $f_{\rm esc} \propto (1+z)^4$).

Ocvirk et al. (2016) find that filaments are sheathed with a layer of hotter gas surrounding a cooler core; the lower temperature and higher density in these filamentary cores results in them having a higher neutral fraction, but they do not find any self-shielding effects (that is, all parts of the filaments observe a UV flux equal to that of the background level). Regions with an overdensity factor of 100 or more, $\rho \geq 100\langle p \rangle$, but which do not contain an ionizing source, do self-shield. This is a slightly more nuanced, but nontheless inside-out, scenario.

O'Shea et al. (2015) find that the faint end of the UV luminosity function flattens out, rather than increasing steeply. That is, they do not find that there are many faint galaxies, emitting ionizing photons, which would be below our current observational detection limits. Such a situation would appear to require other sources of ionizing photons to drive reionization at higher redshifts. Further, this is a robust result from their simulations, in that,

> *... the flattening of the luminosity function is not a result of limited numerical resolution but is caused by the suppression of star formation from radiative and supernova feedback effects in haloes with masses* $M_{\rm vir} \lesssim 2 \times 10^8 M_\odot$.

However, Livermore et al. (2017), from observational data, find strong statistical evidence against any such flattening of the faint end of the UV luminosity function. Rather, they claim that there is observational support for extension of the steep luminosity function, such that it is consistent with the number of faint galaxies necessary for reionization. This tension between observational and numerical results is yet to be resolved.

Hutter et al. (2017) also noted the current conflict in predictions of the *escape fraction*, $f_{\rm esc}$; that is, the fraction of ionizing photons produced by a galaxy which are able to escape their local environment, the interstellar medium (ISM), to reach the IGM and thus power or maintain reionization. Paardekooper et al. (2015) analyze escape fractions in a large number of proto-galaxies forming during the EoR. They find that the escape of ionizing photons is a highly anisotropic process, with typically only a few lines of sight through which photons can escape, particularly for the highest escape fractions. For galaxies with high escape fractions ($\sim 50\%$), there can be many sightlines along which ionizing photons cannot escape, and are thus not visible. They

therefore warn that large samples of potential reionization sources are necessary if we are to observationally constrain the escape fraction. It is also suggested that constant ionizing emissivities, or ones which scale with their host halo mass, as are often used in numerical simulations, are unrealistic. Their simulations produce significant ionizing photons from the lightest haloes, which have the highest escape fractions, but which are below the mass-resolution limit of contemporary reionization simulations. These factors combined could have a significant impact on the topology of reionization as produced in simulations, and also calls into question observational estimates of $f_{\rm esc}$.

More recent work has focussed on incorporating very low mass haloes ($M_H \lesssim 10^8 M_\odot$), or minihaloes, and assessing their importance for reionization. Both Chen et al. (2017) and Kimm et al. (2017) include high escape-fraction minihaloes in relatively small, high-resolution simulations. They are in agreement that minihaloes dominate the earliest stages of reionization, with high-mass haloes ($M_H \gtrsim 10^9 M_\odot$) becoming dominant around $z \approx 10$. However, Chen et al. (2017) ultimately conclude that, of the non-high mass haloes, minihaloes are the primary contributor to reionization, whereas Kimm et al. (2017) find that the main driver is $M_H \gtrsim 10^8 M_\odot$ haloes. Chen et al. (2017) attribute this conflict to different escape fractions having been assumed for haloes in the range $10^8 \lesssim M_H \lesssim 10^9 M_\odot$, their values being approximately an order of magnitude lower.

Thus, while numerical results are generally in agreement that previously-neglected low mass haloes must be included in reionization simulations, there is still substantial disagreement about the relative contributions over the mass range. This problem is compounded by the fact that, at the mass resolution required to resolve minihaloes, simulation volumes of greater than $\sim 10^3\,\mathrm{Mpc}^3$ are computationally intractable, but that this is a rather small volume on the scale of reionization.

2 Radiative Transfer Theory

2.1 The Fundamentals

Rybicki and Lightman (1979) provide a thorough but concise introduction to the fundamental theory, as applied to astrophysical phenomena, some of which is reproduced here. Additionally, a similar notation is employed. A key concept in radiative transfer is the specific intensity, I_ν, defined as

$$I_\nu \equiv \frac{\mathrm{d}E}{\mathrm{d}A\,\mathrm{d}t\;\mathrm{d}\Omega\;\mathrm{d}\nu}, \tag{2.1}$$

that is, the energy $\mathrm{d}E$ passing through an area $\mathrm{d}A$ (normal to the direction of the ray) into a solid angle $\mathrm{d}\Omega$ in a time $\mathrm{d}t$, where the photons have frequencies between ν and $\nu + \mathrm{d}\nu$. When passing through matter, energy may be added or subtracted from a ray due to emission and absorption, respectively. For the case of emission we define the emissivity,

$$j_\nu \equiv \frac{\mathrm{d}E}{\mathrm{d}V\,\mathrm{d}t\;\mathrm{d}\Omega\;\mathrm{d}\nu}, \tag{2.2}$$

which is the energy emitted by a volume element $\mathrm{d}V$ in the frequency range $\mathrm{d}\nu$ into a solid angle $\mathrm{d}\Omega$ during a time $\mathrm{d}t$. After travelling a distance $\mathrm{d}s$, this leads us to a change in the specific intensity I_ν of

$$\mathrm{d}I_\nu = j_\nu\,\mathrm{d}s. \tag{2.3}$$

We define the absorption coefficient α_ν similarly,

$$\mathrm{d}I_\nu = -\alpha_\nu\,I_\nu\,\mathrm{d}s. \tag{2.4}$$

We can understand the above law for absorption in terms of a number density n of absorbing particles, each with a cross-section (i.e. effective absorbing area), σ_ν, that is frequency-dependent. The number of absorbers in a cylindrical volume $\mathrm{d}V = \mathrm{d}A\,\mathrm{d}s$ is $N = n\,\mathrm{d}A\,\mathrm{d}s$, hence the total absorption area is $N\sigma_\nu = \sigma_\nu\,n\,\mathrm{d}A\,\mathrm{d}s$, and so the energy absorbed from the beam is

$$\mathrm{d}E = \mathrm{d}I_\nu\;\mathrm{d}A\;\mathrm{d}\Omega\;\mathrm{d}t\;\mathrm{d}\nu = -I_\nu \cdot N\sigma_\nu \cdot \mathrm{d}\Omega\;\mathrm{d}t\;\mathrm{d}\nu. \tag{2.5}$$

Cancelling terms and comparing to Eq. (2.4) gives us

$$\alpha_\nu = n\sigma_\nu. \tag{2.6}$$

Note it has been assumed that the absorbers are uniformly randomly distributed and independent. In astrophysical situations we can reasonably suppose both of these hold.

With the above definitions in mind, one can construct the equation of radiative transfer in one dimension,

$$\frac{\mathrm{d}I_\nu}{\mathrm{d}s} = -\alpha_\nu\,I_\nu + j_\nu. \tag{2.7}$$

This can be solved in generality, but first consider the case of absorption only, which will prove a useful approximation in Section 2.3. For zero emissivity we have

$$\frac{\mathrm{d}I_\nu}{\mathrm{d}s} = -\alpha_\nu I_\nu, \tag{2.8}$$

which has the solution,

$$I_\nu(s) = I_\nu(s_0) \exp\left[-\int_{s_0}^{s} \alpha_\nu(s')\ \mathrm{d}s'\right], \tag{2.9}$$

which may be written more compactly as,

$$I_\nu(s) = I_\nu(s_0)\, e^{-\tau_\nu(s)}. \tag{2.10}$$

We have introduced another useful term, the optical depth, defined by

$$\mathrm{d}\tau_\nu = \alpha_\nu\, \mathrm{d}s, \tag{2.11}$$

and hence,

$$\tau_\nu(s) = \int_{s_0}^{s} \alpha_\nu(s')\ \mathrm{d}s'. \tag{2.12}$$

The phrase *optically thick* applies when $\tau_\nu > 1$, and denotes that a photon of frequency ν will typically be absorbed when travelling through the medium. The optically thin regime, $\tau_\nu < 1$, is the opposite: the photon will typically traverse the entire integrated path without being absorbed.

To find the general, formal solution to Eq. (2.7), first rewrite it in terms of the optical depth,

$$\frac{\mathrm{d}I_\nu}{\mathrm{d}\tau_\nu} = -I_\nu + S_\nu, \tag{2.13}$$

where both sides have been divided through by α_ν, and $S_\nu \equiv j_\nu / \alpha_\nu$ is the *source function*, the ratio of the emission to absorption coefficients. The solution to Eq. (2.13) is then (Rybicki and Lightman, 1979),

$$I_\nu(\tau_\nu) = I_\nu(0)\, e^{-\tau_\nu} + \int_0^{\tau_\nu} e^{-(\tau_\nu - \tau'_\nu)}\, S_\nu(\tau'_\nu)\ \mathrm{d}\tau'_\nu, \tag{2.14}$$

i.e. the initial intensity reduced by absorption plus the integrated source function along the path, also reduced by absorption.

2.2 The On-the-Spot Approximation

It is sensible at this point to introduce a common approximation made in both analytic and numerical studies, the so-called on-the-spot (OTS) approximation. A recombination to the ground state releases a photon with sufficient energy to ionize another particle; if we suppose that this photon is absorbed very close to its point of origin (or 'on-the-spot'), then all recombinations *to the ground state* are effectively cancelled out. For convenience let us now define the *case B* recombination rate (as compared to the *case A* rate, where all recombinations contribute),

$$\alpha_B \equiv \alpha_A - \alpha_1 \equiv \sum_{n=1}^{\infty} \alpha_n - \alpha_1 = \sum_{n=2}^{\infty} \alpha_n, \tag{2.15}$$

where α_n denotes a recombination to the nth energy level. There is something of an implicit assumption here that recombinations to higher energy levels produce negligible ionizing photons, which is not unreasonable: such photons can ionize only atoms in excited states, which are rare. Further, the diffuse radiation photons have $\nu \approx \nu_0$, the ionization threshold frequency (Osterbrock, 1989). Under the OTS approximation we can hence completely ignore the contribution of recombination photons to the ionizing radiation — quite a substantial simplification.

Numerically, it is particularly valuable in *ray tracing* schemes (see Chapter 3, Section 2), where the computational cost increases with the number of sources. Treating recombinations in full implies that every volume element in the simulation may become a source! It is worth noting that, while potentially valid in the optically thick regime, using case B rates in an optically thin medium is clearly erroneous. In an environment with $\tau \ll 1$ we would be better to use the case A rate, and assume instead that all recombination photons escape the medium. As a result, sometimes case A and case B approximations are referred to, rather than specifically OTS. Some codes, e.g. URCHIN (Altay and Theuns, 2013), estimate the local optical depth and switch between α_A and α_B accordingly.

The OTS approximation is most simply applied in the case of a hydrogen-only gas, as hydrogen has only one ionized state. Often we wish to include helium in cosmological ionization studies; its two ionization states — coupled with the fact that hydrogen is also present — make the 'simple' OTS approximation somewhat harder to deal with (for yet heavier elements the approximation is not used). We could assume that recombinations

to the ground state of species A only go on to photoionize particles of species A, and simply use the case B rates in all cases. This is an uncoupled OTS approximation. Friedrich et al. (2012) introduced a coupled OTS approximation, where recombinations of helium contribute to *both* hydrogen and helium ionizations. They also consider ionizations resulting from de-excitations in helium due to recombinations to (all) states $n \geq 2$. In this case the rates are substantially more complex than simply using each species' value of α_B (see Friedrich et al. (2012) Table 1). A comparison of the two OTS schemes to the results of the CLOUDY code (version 08.00), which employs full transfer of recombination photons, suggests that coupled OTS is a significant improvement.

The validity of the OTS approximation is yet to be fully investigated, though some studies have made progress in this area. Ritzerveld (2005) showed, analytically, that the inclusion of diffuse radiation has a profound effect in all but the simplest HI distributions (e.g. growth of an ionization bubble in an isotropic medium). Cantalupo and Porciani (2011) implemented full transport of recombination photons (from hydrogen and helium) in the code RADAMESH and found the inclusion of HeII and HeIII recombination emission to be important, even in simple hydrogen-helium media where one is only interested in the hydrogen component. Paardekooper (2010) made improvements to the SIMPLEX code (see Section 4.2) and found that the OTS approximation is likely not sufficient for galaxy-scale simulations, particularly when it comes to star formation; however, it was found to be quite accurate for large scale *reionization* simulations.

2.3 The 3D Equation of Radiative Transfer

The three dimensional cosmological radiative transfer equation is given by (see Appendix A of Gnedin and Ostriker (1997) for a derivation),

$$\frac{1}{c}\frac{\partial I_\nu}{\partial t} + \frac{\hat{\mathbf{n}}\cdot\nabla I_\nu}{\bar{a}} - \frac{H}{c}\left(\nu\frac{\partial I_\nu}{\partial \nu} - 3I_\nu\right) = j_\nu - \alpha_\nu I_\nu, \tag{2.16}$$

where $\hat{\mathbf{n}}$ is the unit vector along the direction of propagation of the ray; $\bar{a} = a/a_e$ is the ratio of the scale factors at photon emission (a_e) and a time t (a); $H = \dot{a}/a$ is the time-dependent Hubble constant; and all other terms are as above. Time dependence is introduced in the first term; the second term is just the 3D extension to the left-hand side (LHS) of Eq. (2.7); the third term is due to cosmic expansion. As noted by Abel et al. (1999), the above is simply the usual 3D equation of radiative transfer with two

additions, and is in fact not so different from Eq. (2.7): the factor of $1/\bar{a}$ in the second term accounts for changes in the proper path length due to cosmic expansion, and the third term accounts for cosmological redshift and dilution.

Equation (2.16) covers a large parameter space and so is currently impractical for direct numerical integration. To mitigate this fact some approximations are necessary. They have been employed in a variety of radiative transfer codes (Norman et al., 1998; Abel et al., 1999; Altay et al., 2008; Cantalupo and Porciani, 2011), and are covered below.

Suppose we have a simulation volume of side-length L. Comparing the second and third terms in Eq. (2.16), we may ignore the latter provided that $L \ll c/H$, i.e. that the box size is much smaller than the horizon (Norman et al., 1998; Abel et al., 1999). Furthermore, if a photon is emitted at a time t at one end of the box, it will reach the other at a time $\sim t + L/c$, and so $\bar{a} \sim \left(\frac{t+L/c}{t}\right)^{\eta} \sim 1 + \eta L/ct$, where η is the logarithmic expansion factor ($\eta = 2/3$ for a flat, matter-dominated universe). Supposing also that $L \ll ct$, we have $\bar{a} \approx 1$ (Norman et al., 1998). This leaves us with the non-cosmological form of Eq. (2.16),

$$\frac{1}{c}\frac{\partial I_\nu}{\partial t} + \hat{\mathbf{n}} \cdot \nabla I_\nu = j_\nu - \alpha_\nu I_\nu, \tag{2.17}$$

though note that the absorption and photoionization cross-sections must be appropriately redshifted. In the case of constant absorption and emission coefficients the time derivative may be dropped, and we end with a form that is rather similar to Eq. (2.7),

$$\hat{\mathbf{n}} \cdot \nabla I_\nu = j_\nu - \alpha_\nu I_\nu. \tag{2.18}$$

In general this approximation is sufficient, though it breaks down close to sources (or for very bright sources), allowing ionization front velocities greater than the speed of light (Altay et al., 2008; Cantalupo and Porciani, 2011). A simple solution in the ray tracing scheme is to prevent rays from travelling a distance greater than $ct_{\rm on}$, where $t_{\rm on}$ is the length of time for which the source has been active.

If we trace the properties of the medium along rays then Eq. (2.18) reduces to the 1D case of Eq. (2.7), with s being along the path of the ray. For our numerical purposes, then, the 1D solution is generally sufficient.

2.4 The Ionization Equations

Though only hydrogen is considered here, the same arguments may be applied to helium. The relative ratios of neutral and ionized hydrogen are denoted by $x_{\mathrm{HI}} \equiv n_{\mathrm{HI}}/n_{\mathrm{H}}$ and $x_{\mathrm{HII}} \equiv n_{\mathrm{HII}}/n_{\mathrm{H}}$, respectively, and it is required that $x_{\mathrm{HI}} + x_{\mathrm{HII}} = 1$. We consider three processes for evolving the ionization fractions:

1. Photoionization (Γ),
2. Recombination (α),
3. Collisional Ionization (γ).

The bracketed symbol is the one associated with the rate of each process. Adopting a somewhat compacted notation, the ionization evolution equations are (Altay et al., 2008)

$$\frac{\mathrm{d}x_{\mathrm{HI}}}{\mathrm{d}t} = -G_{\mathrm{HI}}\, x_{\mathrm{HI}} + R_{\mathrm{HII}}\, x_{\mathrm{HII}}, \tag{2.19a}$$

$$\frac{\mathrm{d}x_{\mathrm{HII}}}{\mathrm{d}t} = G_{\mathrm{HI}}\, x_{\mathrm{HI}} - R_{\mathrm{HII}}\, x_{\mathrm{HII}}, \tag{2.19b}$$

where the ionization terms have been grouped as $G_A \equiv \Gamma_A + \gamma_A\, n_e$, for some photoabsorbing species A, and the recombination terms grouped as $R_I = \alpha_I\, n_e$, where n_e is the number density of electrons (equal to the total number density of ionized species, in this simple case n_{HII}), for some photoionized species I. Γ_A gives the ionization rate per unit time per unit volume per particle of species A,

$$\Gamma_A = \int_\Omega \int_{\nu_{A,0}}^{\infty} \frac{I_\nu \sigma_{A,\nu}}{h\nu}\, d\nu\, d\Omega, \tag{2.20}$$

sometimes (e.g. Osterbrock (1989)) defined in terms of the mean intensity of radiation, J_ν, as

$$\Gamma_A = \int_{\nu_{A,0}}^{\infty} \frac{4\pi J_\nu\, \sigma_{A,\nu}}{h\nu}\, d\nu, \tag{2.21}$$

where $\nu_{A,0}$ is the ionization threshold for species A. For the frequency- and species-dependent cross-sections, as well as the temperature- and species-dependent recombination (α) and collisional ionization (γ) rates, fits to atomic data are used. Table 1 of both Pawlik and Schaye (2011) and Altay et al. (2008) list some common sources for this data.

A component not thus far discussed is *secondary ionizations*. When a highly energetic

(e.g. X-ray) photon ionizes a particle, a fast non-thermal-equilibrium electron is released; this electron may go on to (secondarily) collisionally ionize other neutral particles. Full treatment of this phenomenon is challenging, generally done (Friedrich et al., 2012; Feng et al., 2013) using fits to the results of Monte Carlo computations, for example those of Shull and van Steenberg (1985) and Furlanetto and Stoever (2010).

2.5 The Temperature Equation

The given form of the temperature equation varies by author, often being written instead in terms of internal energy (Pawlik and Schaye, 2011; Cantalupo and Porciani, 2011). Here is given the form as used in the radiative transfer code SPHRAY (Altay et al., 2008),

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{2}{3nk_B}\left(\mathcal{H} - \Lambda\right) - \frac{T}{n}\frac{\mathrm{d}n}{\mathrm{d}t}\,, \tag{2.22}$$

where n is the number of free particles per unit volume. $\mathcal{H}$ is the photoheating term, with units of energy per unit volume per unit time, and represents the kinetic energy added to the system from photoionized electrons. In general, the photoheating rate is given by (Pawlik and Schaye, 2011),

$$\mathcal{H}_A = \int_\Omega \int_{\nu_{A,0}}^{\infty} \frac{I_\nu \sigma_{A,\nu}}{h\nu}(h\nu - h\nu_{A,0})\ \mathrm{d}\nu\ \mathrm{d}\Omega, \tag{2.23}$$

notably similar to Eq. (2.20), but the integrand has been multiplied by the excess energy over the ionization threshold, i.e. the kinetic energy imparted to the photoionized electron. It is assumed that this energy is rapidly transferred to other gas particles.

The cooling term, Λ, is less straightforward. The commonly used contributing physical processes are:

1. Collisional ionization cooling (ζ),
2. Collisional excitation cooling (ψ),
3. Recombination cooling (η),
4. Bremsstrahlung cooling (β),
5. Compton (heating and) cooling (χ).

The above terms are all functions of temperature, and combine together as (Altay et al., 2008)

$$\Lambda = \beta n_{\mathrm{HII}} n_e + \chi n_e + \sum_I \eta_I n_I n_e + \sum_A \left(\zeta_A + \psi_A\right) n_A n_e, \tag{2.24}$$

where once again A denotes a photoabsorbing species and I a photoionized one. Note that in the case of helium, the first term above becomes $\beta(n_{\mathrm{HII}} + n_{\mathrm{HeII}} + 4n_{\mathrm{HeIII}})n_e$ (Cen, 1992). As in the case of cross-sections and other rates, the above are generally based on known functions of temperature. See again Table 1 of Pawlik and Schaye (2011) and Table 2 of Altay et al. (2008) for examples.

3 Numerical Simulations in Astrophysics

In Section 4, computation methods to solve the above equations are discussed. However, in order to be of practical value, such methods must be compatible with the methods used for numerical simulation of gravity and hydrodynamics, both of which are dominant processes on the cosmological scale. In particular, radiative transfer must be performed on a fluid field of some form, and so this section begins with a discussion of the fundamentals of SPH and, briefly, adaptive mesh refinement (AMR). Following this, more modern techniques involving unstructured meshes, or mesh-free formalisms, are outlined.

As noted above, on cosmological scales — and unlike fluid dynamics in many other fields — the self-gravity of the fluid is of critical importance; for this reason, discussion begins with modelling N-body dynamics.

3.1 N-body Dynamics

N-body dynamics is a much-studied subject with a wealth of literature. It is also somewhat outside the scope of this document; the summary below draws predominantly from Dehnen and Read (2011), Bagla (2005) and Heggie and Hut (2003), and interested readers are direct towards those references, and others therein.

There are two categories of N-body system: collisional and collisionless. Collisional systems are those with dynamical time scales much shorter than their age, where the net result of all close two-body interactions significantly affects the overall evolution. (It is these such two-body encounters that are meant by the term 'collisions'.) In a collisionless system the long-term effects of the two-body close encounters are negligible, and we can approximate the sum of the individual point-mass gravitational potentials by a single mean potential $\Phi(\mathbf{r}, t)$. This potential obeys Poisson's Equation,

$$\nabla^2\Phi(\mathbf{r}, t) = 4\pi G\rho(\mathbf{r}, t). \tag{2.25}$$

In the cosmological context, non-interacting dark matter falls into this collisionless regime, making it the relevant one for our purposes. The Hamiltonian for the particles may be constructed, and depends in part on the gravitational potential. To approximate isotropy and homogeneity in the universe it is common to simulate a box of side-length L and enforce periodic boundary conditions. In this case, the form of the potential includes a sum over all *images* (copies of the simulation volume).

The actual force computation between N bodies, if done as a direct summation, involves $O(N^2)$ computations. This is generally far too costly, so approximations have been developed. See Dehnen and Read (2011) for a very complete selection; here only a few are noted.

In the **Tree Method** (Barnes and Hut, 1986) we begin with a cube (the *root node*) containing all simulation particles, and recursively sub-divide this node into octants, with each octant being a node itself (this kind of structure is known as an *octree*). A common limit for the sub-division is that a node contains only one particle, in which case it is known as a *leaf*. At large distances, the effect of all the particles contained within a node on a given body may be estimated from the multipole moments of the node — the simplest monopole moment requires only knowledge of the node's total mass and centre of mass (CoM) position vector. Key to this method, then, is determining when the multipole approximation is acceptable. Most simply, a cell-opening criterion, θ, is set. To calculate the force on a particle p consider a node of side-length l, with its CoM a distance D from p. If $l/D \leq \theta$ the approximation is accepted and the cumulative force on p increased; if $l/D > \theta$ we reject the cell, 'open' it, and perform the same test on its child nodes, proceeding recursively. This algorithm scales as $O(N \log N)$ and permits arbitrary accuracy, tending to the direct-sum solution as $\theta \to 0$.

The **particle mesh (PM)** method makes use of the fact that Poisson's Equation (2.25) becomes simply $\mathbf{k}^2\hat{\Phi}(\mathbf{k}, t) = -4\pi G\hat{\rho}(\mathbf{k}, t)$ in Fourier space. A mesh is imposed over the particle field, and the density found at the vertices. Then the Poisson Equation is solved, via a Fast Fourier Transform (FFT), to give Φ at each vertex. Interpolating between the vertices gives an estimate of the potential for any given particle (in fact, at any arbitrary point). For N_g grid points, the FFT algorithm scales as $O(N_g \log N_g)$, and for N particles the interpolation scales as $O(N)$. However, the computed forces at pair separations less than several grid spacings are a poor approximation to the inverse square law (Bertschinger, 1998).

The **particle-particle particle-mesh ($\mathbf{P^3M}$)** method attempts to compensate for PM's underestimation of short-range forces by calculating direct-sum forces for near-neighbours. A near-neighbour is any particle within ~ 2 mesh grid-lengths, and these corrective terms are simply added to the total force. PM actually underestimates the forces to distances greater than is usually corrected for in P^3M; additionally, the short-range corrections assume the long-range forces are isotropic, which is generally not the case.

The **TreePM** method, used by e.g. the cosmological SPH code GADGET-2, splits the gravitational force into long- and short-range components, based on some tuneable splitting-scale parameter, r_s. The long-range force is calculated using the PM method, and the short-range force by a tree code.

3.2 Adaptive Mesh Refinement

While gravitational effects are of utmost importance on many astrophysical scales, proper modelling of hydrodynamics (gas physics) is often also essential. Hydrodynamic solvers are typically divided into two categories: Eulerian and Lagrangian. All grid-based codes are Eulerian in nature: they follow the dynamics of the fluid through fixed volume elements, e.g. using a grid of equal-sized cells. A key problem with this naïve equal-cell grid method is that, in order to sufficiently resolve areas of interest (e.g. those of high density), the entire simulation volume must be sub-divided into a large number of very fine regions. This rapidly becomes computationally intractable, and for this reason adaptive mesh refinement (AMR) was developed (Berger and Colella, 1989), wherein much of the grid is coarse and cells are refined (sub-divided) from the *root* (coarsest) level only when needed.

In astrophysics, several gravity hydrodynamics AMR codes have been developed and made publicly available, including RAMSES[5] (Teyssier, 2002), FLASH[6] (Fryxell et al., 2000), GAMER[7] (Schive et al., 2010) and ENZO[8]. Of these, RAMSES, FLASH and ENZO in particular are large, mature projects containing many modules for additional physics, including radiative transport and magnetohydrodynamics.

Several methods of refinement exist. **Block- or patch-based AMR**, as used by

[5] http://www.itp.uzh.ch/~teyssier/Site/RAMSES.html
[6] http://flash.uchicago.edu/site/flashcode/
[7] http://iccs.lbl.gov/research/isaac/GAMER_Framework.html
[8] http://enzo-project.org/

e.g. ENZO, is the simplest. Originally introduced by Berger and Colella (1989), it refines rectangular regions as a whole. This reduces the complexity of dealing with resolution discontinuities between refinement levels, but the variable (and often large) patch size makes efficient parallelization and data management difficult (Schive et al., 2010).

Cell-based AMR, as used by e.g. RAMSES, refines the grid on a cell-by-cell basis. As a result, the refinement geometry is much more flexible and better represents the underlying properties of the medium. However, the highly irregular structure formed requires yet more advanced data management, and solving across boundaries at different resolutions becomes both more common and more complex. Interpolation is often required to estimate the boundary conditions of a particular cell.

Hierarchical patch-based AMR, as used by e.g. FLASH, is similar to the block-based method, but all patches are required to contain the same number of cells. It is in some sense the most inflexible method described, and is likely to lead to larger areas of refinement. Its simplicity, though, is its key advantage: the data structure and resolution discontinuities are much less complex due to the regular sub-divisions, meaning efficient parallelization is relatively straightforward.

All Eulerian codes (AMR or not) attempt to solve the Euler equations, which in their conservative form (and including gravity) are (Teyssier, 2002, for example),

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \tag{2.26a}$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \times \mathbf{u}) + \nabla p = -\rho \nabla \Phi, \tag{2.26b}$$

$$\frac{\partial (\rho e)}{\partial t} + \nabla \cdot \left[\left(e + \frac{p}{\rho} \right) \rho \mathbf{u} \right] = -\rho \mathbf{u} \cdot \nabla \Phi, \tag{2.26c}$$

where ρ is the mass density, Φ the gravitational potential, $\mathbf{u}$ the bulk fluid velocity, e the specific total energy and p the thermal pressure. Note that Eq. (2.26c) explicitly conserves the total fluid energy only if gravity is ignored. As before, Φ obeys Poisson's Equation (2.25). The above equations are discretized over the grid (presenting us with a Riemann problem) and solved using a finite-volume method. Typically, a total-variation diminishing (TVD) scheme or one based on Godunov's method is implemented (Bertschinger, 1998).

To solve for the gravitational potential it is common to finite-difference Eq. (2.25) and solve via an FFT over the root level. Since at any given level of refinement beyond

the root level the entire simulation volume is likely not resolved, this cannot be done at any higher resolution. It is common to then employ a relaxation method, whereby the potential in a refined grid cell is iteratively updated using its previous value, the previous value of neighbouring cells, its density, and a relaxation parameter (Teyssier, 2002; Schive et al., 2010).

3.3 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) is a Lagrangian technique, that is, each node (particle) of the simulation follows the motion of an associated material particle. Several reviews on the topic exist (Monaghan (1992, 2005); Springel (2010b) to name but a few), the basics of which are discussed in this section. In SPH, particles are effectively interpolation points, and the value of some scalar field A at position $\mathbf{r}$ is estimated as

$$\tilde{A}(\mathbf{r}) = \int A(\mathbf{r}')\, W(\mathbf{r} - \mathbf{r}', h)\, \mathrm{d}\mathbf{r}'. \tag{2.27}$$

The function W is known as the *SPH kernel*, which effectively smooths the properties of the field over a characteristic length h, approximating a δ-function in the limit $h \to 0$. W is normalized to one, and should be both symmetric and at least twice-differentiable — an example is given in Eq. (2.31).

To discretize the field we first rewrite Eq. (2.27) as

$$\tilde{A}(\mathbf{r}) = \int \frac{A(\mathbf{r}')}{\rho(\mathbf{r}')}\, W(\mathbf{r} - \mathbf{r}',\, h)\, \rho(\mathbf{r}')\, \mathrm{d}\mathbf{r}', \tag{2.28}$$

where ρ is density, and hence a differential mass element is $\mathrm{d}m(\mathbf{r}') = \rho(\mathbf{r}')\, \mathrm{d}\mathbf{r}'$. Now, consider the fluid to be composed of particles with masses m_i and densities ρ_i, at positions $\mathbf{r}_i$ and with smoothing lengths h_i. In the continuous limit, Eq. (2.27), the mass of each element is the above $\mathrm{d}m(\mathbf{r}')$. Then, taking the field to be known at the set of discrete points $\mathbf{r}_i$ with masses m_i, we may approximate Eq. (2.27) as a summation over these known mass elements,

$$\tilde{A}(\mathbf{r}) \simeq \sum_i \frac{m_i}{\rho_i}\, A_i\, W(\mathbf{r} - \mathbf{r}_i,\, h_i), \tag{2.29}$$

which clearly is defined everywhere, not just at the locations of the known particles. Formally this sum is over all particles; however, the chosen form of W in general has

compact support, vanishing within a finite distance, h (a property which Eq. (2.31) shares).

It may be that all smoothing lengths are equal, $h_i \equiv h \,\forall\, i$; more commonly, however, variable smoothing lengths $h_i \neq h_j$ are chosen (again, Eq. (2.31) shares this property). Typically only a spatial dependence is used, such that the number of near-neighbours (that is, the number of other particles whose smoothing volumes intersect any given particle) is nearly or exactly constant. However, GADGET-2 employs a slightly modified strategy, as is noted below in Eq. (2.36). Defining $h_i \equiv h(\mathbf{r}_i)$ and $W_{ij}(h) \equiv W(|\mathbf{r}_i - \mathbf{r}_j|, h)$ gives an estimate of the density at position $\mathbf{r}_i$ as

$$\rho_i = \sum_{j=1}^{N} m_j \, W_{ij}(h_j). \tag{2.30}$$

A common choice for W is the cubic spline (given by e.g. Springel (2005) for the cosmological SPH code GADGET-2),

$$W(r,h) = \frac{8}{\pi h^3} \begin{cases} 1 - 6\left(\frac{r}{h}\right)^2 + 6\left(\frac{r}{h}\right)^3 & 0 \leq r \leq \frac{h}{2} \\ 2\left(1 - \frac{r}{h}\right)^3 & \frac{h}{2} < r \leq h \\ 0 & r > h \end{cases} \tag{2.31}$$

Springel (2010b) review an elegant Lagrangian-based derivation of the SPH equations of motion, first given by Springel and Hernquist (2002), which is outlined here. The Euler equations for an inviscid (no viscosity) flow may be derived from the Lagrangian,

$$L = \int \rho \left(\frac{\mathbf{v}^2}{2} - u \right) \mathrm{d}V, \tag{2.32}$$

where u is thermal energy per unit mass. The discretized form is,

$$L_\mathrm{d} = \sum_i \left(\frac{1}{2} m_i \mathbf{v}_i^2 - m_i u_i \right). \tag{2.33}$$

The pressure of the particles is

$$P_i = A_i \, p_i^\gamma = (\gamma - 1) \rho_i \, u_i, \tag{2.34}$$

and so

$$u_i(\rho_i) = A_i \frac{\rho_i^{\gamma-1}}{\gamma - 1}, \tag{2.35}$$

where A_i is the specific entropy and γ is the adiabatic index. We now set a condition on the smoothing lengths in order to keep the mass in a kernel volume exactly constant,

$$\rho_i h_i^3 = \text{constant}. \tag{2.36}$$

The Euler-Lagrange equation (with partial derivatives with respect to $\dot{\mathbf{r}}_i$ and $\mathbf{r}_i$) gives

$$m_i \frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \frac{P_j}{\rho_j^2} \frac{\partial \rho_j}{\partial \mathbf{r}_i}\,. \tag{2.37}$$

Noting that $\partial\rho_j/\partial\mathbf{r}_i$ signifies the variation in the density with respect to the particle co-ordinate $\mathbf{r}_i$, *including* any variation in h_j this may entail, and differentiating Eq. (2.36) with respect to $\mathbf{r}_i$ yields the SPH equations of motion (Springel, 2010b),

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \left\{ f_i \frac{P_i}{\rho_i^2} \left.\frac{\partial W_{ij}(h_i)}{\partial \mathbf{r}_i}\right|_{h_i} + f_j \frac{P_j}{\rho_j^2} \left.\frac{\partial W_{ij}(h_j)}{\partial \mathbf{r}_i}\right|_{h_j} \right\}, \tag{2.38}$$

where

$$f_i \equiv \left(1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i}\right)^{-1}. \tag{2.39}$$

Only the one above equation (2.38) need be solved, then, for inviscid gas dynamics. In this formulation energy, entropy, and linear and angular momentum are all conserved, provided smoothing lengths are dynamically adapted such that Eq. (2.36) holds. This was not true of prior formulations of SPH.

It is rather straightforward to include gravitational forces in SPH; each particle may be treated as point-like, and the usual N-body techniques applied (see Section 3.1).

3.4 Moving mesh codes

Moving mesh codes represent the state of the art in astrophysical hydrodynamical simulations. They are often described as semi-Lagrangian, or arbitrary Lagrangian-Eulerian, and are designed to display the best characteristics of both SPH and grid-based methods. The first such code to see widespread use was AREPO (Springel, 2010a), which is based on the GADGET-2 code, sharing its gravity solver and paralellization strategy. Given an input set of points, AREPO constructs its mesh as the Voronoi tessellation of

said points. A Voronoi tessellation results in one cell per input point, the *generating points*. Each cell is formed such that all spatial locations within it are closer to that cell's generating point than to any other cell's generating point. (Use of Voronoi tessellation is not a requirement of the moving-mesh method, but it is convenient and conceptually elegant.)

The resulting cells have well-defined, planar boundaries with all neighbouring cells, and finite-volume methods can be applied to compute the hydrodynamic properties, exhibiting Eulerian characteristics. However, the resulting meshes move, or deform, with the motion of the underlying points, exhibiting Lagrangian behaviour.

More recently, the concept has been further extended to so-scalled 'meshless' methods. Put simply, they are similar to moving mesh methods, but the boundaries between cells are smoothed, according to some kernel function, rather than sharply defined. A notable example within astrophysics is GIZMO (Hopkins, 2015), also based on the GADGET-2 codebase.

4 Numerical Radiative Transfer

This section gives an outline of the techniques currently employed to solve for the ionization and temperature states of a medium. The methods described here may in general be applied to any of the techniques presented in Section 3 for simulating gas hydrodynamics and self-gravity. Due to its common use within the context of cosmological-scale gravity-hydrodynamics simulations, and the comparative dearth of codes catering to that need, particular attention is given to solutions and codes which may be readily applied to SPH datasets. This is also relevant for both Chapters 4 and 5; the GRACE library was initially conceived with SPH inputs in mind, and TARANIS is at present only able to process SPH datasets. Section 4.3 also motivates ray tracing as the method chosen for TARANIS, and hence the development of GRACE.

4.1 Moment Methods

First suggested as a means of including radiative transfer in cosmological hydrodynamics simulations by Norman et al. (1998), the angular dimensions of the RT equation are reduced (typically) by taking its first and second angular moments. The resulting equations are a set of conservation laws, not dissimilar to the Euler equations (2.26)

involved in solving hydrodynamical problems, and hence coupling moment methods to grid-based hydrodynamics codes is relatively straightforward (Commerçon et al., 2011; Rosdahl et al., 2013). These equations can be intuitively thought of as describing the radiation field in terms of the net motion of all photons crossing a given volume element. Additionally, in moment methods the computational cost does not scale with the number of sources, only the number of volume elements.

The loss of directionality has some drawbacks, however: namely, the radiation is generally transported in a rather diffuse manner, preventing proper modelling of shadowing. In the optically thick regime, where the photon mean free path is short, this diffusive nature is a good description of the physics; in the optically thin regime it is often far less so. A recent moment-based code, RAMSES-RT (Rosdahl et al., 2013) — a radiation solver added and coupled to the hydrodynamics in the AMR code RAMSES (Teyssier, 2002) — is aimed more toward galaxy-scale simulations, as this is where it performs best. The authors suggest instead use of the ATON GPU-accelerated code (Aubert and Teyssier, 2008, 2010) for cosmological reionization problems; they also note that it is rather difficult to couple ATON's fixed Cartesian grid-based solver with the AMR hydrodynamic solver of RAMSES.

Taking the first two moments (zeroth and first) of Eq. (2.17) gives (Aubert and Teyssier, 2008, 2010; Rosdahl et al., 2013)

$$\frac{\partial N_\nu}{\partial t} + \nabla \cdot \mathbf{F}_\nu = -\sum_A^{\text{HI,HeI,HeII}} \alpha_{\nu A}\, c\, N_\nu + S_\nu, \tag{2.40a}$$

$$\frac{\partial \mathbf{F}_\nu}{\partial t} + c^2 \nabla \cdot \mathbb{P}_\nu = -\sum_A^{\text{HI,HeI,HeII}} \alpha_{\nu A}\, c\, \mathbf{F}_\nu, \tag{2.40b}$$

where c is the speed of light; $\alpha_{\nu A}$ is the absorption coefficient of species A at frequency ν; N_ν is the photon number density; $\mathbf{F}_\nu$ is the photon flux; $S_\nu \equiv \dot{N}_\nu^* + \dot{N}_\nu^{\text{rec}}$ is the source term, consisting of photon-producing sources (e.g. stars), $\dot{N}_\nu^*$, and recombination radiation, $\dot{N}_\nu^{\text{rec}}$ (it is a distinct advantage that recombinations can be so easily included in moment methods, particularly as the validity of the OTS approximation has not yet been thoroughly investigated); and $\mathbb{P}_\nu$ is the radiative pressure tensor, which in some sense determines the direction of flow of the radiation, and must be given in order to close the set of equations. Multiple tensors are available for this purpose, each with various advantages and disadvantages.

In the simplest case of **flux-limited diffusion (FLD)**, only the zeroth-order moment of the RT equation is taken. The equations are closed by a diffusion term which allows radiation to flow in the opposite direction to the radiative energy gradient; the flux limiter is chosen to limit the flux to that physically possible given the finite speed of light (Rosdahl et al., 2013). While this allows for both optically thick (diffusion limit) and optically thin (free-streaming limit) approximations (Bodenheimer et al., 1990; Kuiper et al., 2010), it can cause issues in transition, or under a combination of both. For example, radiation will wrap around opaque obstacles rather than producing well-defined shadows (González et al., 2007; Roth and Kasen, 2015), potentially having unphysical effects on should-be shadowed material (Kuiper and Klessen, 2013). Most applications of FLD occur in the valid, optically thick regime — for example, simulations of protostellar collapse (Commerçon et al., 2011; Cunningham et al., 2011; Tomida et al., 2010) — though the approximation has also been adopted for cosmological reionization (e.g. Aubert and Teyssier, 2008; Norman et al., 2015), an optically-thin problem. A major advantage of this method is that the computational cost is independent of the number of sources, a desirable feature for codes aimed at modelling reionization in cosmological volumes, where the number of sources is large ($\gtrsim 10^4$).

An alternative, designed with the optically thin limit in mind, is the **optically thin variable Eddington tensor (OTVET)**. ($\mathbb{P}_\nu$ may be expressed as $N_\nu \mathbb{D}_\nu$, where $\mathbb{D}_\nu$ is known as the *Eddington tensor*.) The Eddington tensor in the OTVET scheme is calculated assuming an optically thin medium between all points and sources. The directionality of the radiation is thus preserved, but along non-optically-thin lines of transport it is likely that some radiation will be advected in the wrong direction. The computational cost is also increased, though the algorithm still scales independently of the number of sources (Gnedin and Abel, 2001).

Lastly we have the **M1 closure relation** (Levermore, 1984; González et al., 2007), as used in e.g. the RAMSES-RT (Rosdahl et al., 2013) and ATON (Aubert and Teyssier, 2008), codes. See also McKinney et al. (2014); Skinner and Ostriker (2013); Vaytet et al. (2011). Unlike the non-local OTVET method, the M1 closure relation is composed only from local terms, but it still retains some directionality. The Eddington tensor in this scheme contains two terms: the first is isotropic, treating radiation in all directions equally; the second represents the free-streaming case, and contains components which are principally aligned with the *local* flux. All regimes between these two extremes —

which are themselves described exactly by the M1 relation — are treated as a linear combination of the two, and are naturally represented somewhat inexactly. The rough approximation of the actual local radiation geometry leads to other spurious effects; for example, the M1 model will treat two distinct sources as one 'average' source (Aubert and Teyssier, 2008), and beam crossings can result in extremely non-physical behaviour. As shown by e.g. Aubert and Teyssier (2008) and Rosdahl et al. (2013), the M1 relation is capable of preserving shadows, though they are not as well-defined as in ray tracing schemes.

4.2 Direct Photon Transfer

For performing radiation transport on particle-like distributions, some authors have developed methods which exchange photons, or packets of photons, directly between neighbouring particles. Two such methods, and their associated codes, are described below. Both have the advantage that, once the simulation is sufficiently saturated with in-flight *photon packets*, the computational time does not scale (or depends only weakly) on the number of sources. Both are also ideal fits for optically thick media, but have to take special care to transport radiation along straight paths in the optically thin case.

SimpleX

The SimpleX code (Ritzerveld and Icke, 2006; Paardekooper et al., 2010) implements photon transfer on an unstructured lattice, which is adaptive based on the properties of the medium; more grid points are present in areas which have higher opacity. To begin, grid points are placed with a point intensity function which is the convolution of a homogeneous Poisson distribution and a (possibly) inhomogeneous function related to the density of the medium. That is,

$$n_{\mathrm{p}}(\mathbf{x}) = \Phi * f[n(\mathbf{x})], \tag{2.41}$$

where $n_{\mathrm{p}}(\mathbf{x})$ is the point intensity function, Φ is the probability function for the Poisson point process, and $f[n(\mathbf{x})]$ is a function of the number density of the medium. The set of points formed in this way are used as the generating points, or nuclei, for a Voronoi tessellation (recall Section 3.4). Delaunay triangulation is then performed: lines join

each nucleus whose associated volumes share a common boundary. These lines, known as Delaunay edges, form a tessellated set of triangles in 2D and tetrahedra in 3D.

Radiation transport occurs along the Delaunay edges, between neighbouring cells. Supposing the photons travel a distance between grid points which gives rise to an optical depth $\Delta\tau$, then the number of photons absorbed by a cell, $N_{\rm abs}$, is related to the number of incoming photons, $N_{\rm in}$, as,

$$N_{\rm abs} = N_{\rm in}\left(1 - e^{-\Delta\tau}\right), \tag{2.42}$$

and so $N_{\rm out} = N_{\rm in}e^{-\Delta\tau}$ photons continue on.

Because the directions photons may travel in are dictated by the Delaunay edges, SimpleX incorporates three different transport modes. For scattering processes, a cell directs photons outward along all edges. Scattering processes and diffuse radiation from recombinations are thus very naturally treated by this method. (For the sake of straightforward comparisons to other codes, Paardekooper et al. (2010) nonetheless employ the OTS approximation in their tests.)

The $N_{\rm out}$ photons not interacting with a cell should, in cases other than the above, continue in the same direction from which they entered the cell. This is in general not possible exactly, so the *ballistic transport* method is employed. Here, outgoing photon packets are divided into equal parts and sent along the d edges most-parallel to the incoming edge, where d is the dimension of the simulation. Photons travelling backwards, i.e. at $> 90^\circ$ with respect to the incoming direction, are explicitly forbidden, with $n < d$ edges chosen instead if necessary.

The ballistic transport method still introduces numerical diffusion, which dominates after approximately five interactions (Paardekooper et al., 2010). To remedy this, SimpleX also has a *direction conserving* transport method. Here, the direction a photon may travel is confined to a solid angle corresponding to its initial direction. Photons travel along only those edges which are within this solid angle. Note that this causes the photon paths to zig-zag, rather than follow true straight lines, requiring a correction to the distance the photons have travelled (since the zig-zag distance is farther than the straight-line distance).

For scattering and diffuse processes the SimpleX method has obvious advantages. For the lower optical depth situations, direction-conserving photon transport must be

imposed to prevent photons from preferentially travelling around areas with low grid point density. In this sense it is rather complimentary to, e.g., ray tracing (see Chapter 3, Section 2) which naturally preserves photon directions but struggles to include diffuse processes in a computationally tractable manner. Further, as the grid of points is unlike that used in the most common gravity and/or hydrodynamic simulations (i.e. some form of regular volumetric particle, be it cubic or spherical) the method cannot take advantage of any data structures that may have been already calculated. The authors do show, however, that the time to construct the triangulation is an order of magnitude lower than the time to complete the radiation transport (Paardekooper et al., 2010).

TRAPHIC

TRAPHIC (Pawlik and Schaye, 2008, 2011) has a somewhat similar method of photon transport to SimpleX, but has been designed to operate directly on an SPH dateset. The angular resolution of incoming and outgoing radiation for each particle is discretized into cones. Photon packets are emitted from source particles and accepted by neighbouring gas particles; the gas particles then re-emit these packets to their neighbours, and so on. Again, this naturally allows for efficient treatment of diffuse radiation and scattering processes. TRAPHIC has a tunable angular resolution for both the incoming and outgoing cones, and similarly to the direction conserving transport method of SimpleX, each photon packet stores its original cone of emission. In the limit of infinitesimal emission and reception cones, this method is essentially identical to ray traced transport of photon packets. Free-streaming (the optically thin case) may be modelled by re-emitting each photon packet into the same solid angle as it was originally emitted.

An important optimization in TRAPHIC is the merging of photon packets. Photons which lie in the same receiving cone of a given particle will be re-emitted as a single packet. This fully constrains the maximum number of photon packets that may simultaneously be emitted by any particle and, hence, the entire simulation, substantially reducing the computation cost relative to maintaining all packets from emission to extinction.

4.3 Ray Traced Radiative Transfer

There are several different schemes, all based on tracing *rays* through the simulation volume. Details of the tracing process itself — that is, determining which volume

elements are intersected by the rays, is discussed in Chapter 3, and *ray tracing* algorithms within the context of astrophysical simulations are covered in Chapter 3, Section 3.4. The focus here is instead on what one does with that information in numerical radiative transfer.

As might be expected, many astrophysical ray tracing codes exist; the majority of these are grid-based, though some exist for SPH fields. In the below, the terms *cell* and *volume element* are used interchangeably, as the concepts apply to both. It is this latter category to which attention will be given, in particular the Monte Carlo code SPHRAY (Altay et al., 2008), which appears to be the only publicly-available radiation transport solver which can operate directly on an SPH dataset — other, grid-based solvers are publicly available, but obviously require an SPH to grid conversion first.

In ray tracing, the fundamental idea is to solve for $I_\nu(\tau_\nu)$ in Eq. (2.17) — or Eq. (2.18) in the time-independent case — along one-dimensional lines, i.e. rays, which are *cast* from sources. Note that we treat I_ν as a function of the optical depth, τ_ν, as in Eqs (2.13)–(2.14). The most direct method — and one which best illustrates the principles — is that of ***long characteristics*** (see Chapter 3, Section 3.4), in which, typically, one ray is cast from each photo-emitting source to each photo-receiving volume element. The tracing process itself will determine the total optical depth along the ray, or some proxy for it. Given the number density of each photoabsorbing species in each volume element, the species' cross-sections, cell temperatures and the emitting flux (or luminosity) of the source(s), one may compute a cell's photoionization, photoheating and cooling rates, allowing the cell to be updated.

Short characteristics is similar (again, see Chapter 3, Section 3.4), but attempts to reduce the number of rays traced. Mellema et al. (2006), for example, use the short characteristics method in C^2-RAY, and notably also introduced an efficient, implicit time-stepping scheme. In their method, a time-averaged (over an integration time step) optical depth is calculated for each cell. This averaged optical depth gives averaged photoionization and heating rates, which in turn give time-averaged ionization fractions, leading to new optical depths; the loop is repeated until convergence in the ionized fractions. This causal approach to solving the equation of radiative transfer introduces some 'seriality' to the code, but is mitigated by the fact that it allows for much larger time-steps (a factor of $\sim 10^3$) to be taken. Accuracy is also particularly good for the monochromatic case (that is, only one frequency of photon in a ray) without

recombinations. This advantage is lost if multi-frequency radiation is introduced, with the method giving poor photon conservation (Mackey, 2012).

Finally, **Monte Carlo** schemes sample the radiation field randomly in both angle and frequency, and typically transport *photon packets*, bundles of multiple photons which are deposited into volume elements. The code CRASH (Ciardi et al., 2001) is an early example of this method, and has since been updated to include: helium and its ionization states, gas temperature evolution and background radiation (Maselli et al., 2003); multi-frequency photon packets (Maselli et al., 2009); a distributed-memory parallel implementation (Partl et al., 2011); and most recently, the ionization states of metals using the photoionization code CLOUDY (Ferland et al., 1998; Graziani et al., 2013). A photon packet is emitted from a source in a random direction, and the energy of the packet is also Monte Carlo sampled from the spectral energy distribution of the source. Naturally, as with any statistical model, (shot) noise is an issue; it can be kept arbitrarily low by increasing the number of rays traced, at the cost of increased computation time.

Transporting photon packets also allows for a reduced communication overhead for massively parallel, distributed-memory machines: once a photon packet reaches a domain boundary, its inherent quantities, such as the number and frequency of remaining photons and its direction of propagation, may simply be passed on to another domain. This is not true of other ray-tracing schemes, where the optical depth at all (discrete) points along a ray is required, implying substantial communication or co-ordination for rays spanning multiple domains. It should be noted that the photon-packet method is not unique to Monte Carlo schemes, being used, for example, in ENZO (Wise and Abel, 2011) and the above-mentioned SIMPLEX (Paardekooper et al., 2010).

The main advantages of the ray tracing scheme are its conceptual simplicity and accuracy. Of all the methods covered in this section, it is typically the most straightforward to implement (efficiency notwithstanding), whilst also attempting to solve the equation of radiative transfer, Eq. (2.17), in the most direct manner. The only, but significant, disadvantage is its computational cost, and the complexity that is introduced in alleviating this cost. The processing time increases linearly in the number of sources, which may be problematic in the many-sourced cosmological context, and also in optically thick environments where any or all volume elements may be emitters. This balance of advantages and disadvantages is the main motivation for pursuing high-performance

ray tracing: that ray tracing has the potential to accurately model all relevant forms of radiation transport, the only (admittedly substantial) hindrance being its computational cost. Just as in computer graphics, ray tracing is the most accurate solution, but is also the most computationally expensive.

In addition to the relative ease with which they may be parallelized, and their high accuracy, ray-tracers offer another advantage: the solvers, i.e. the implemented physics, are easily separated from the tracing. This is particularly obvious in the case of Monte Carlo photon packets, but applies to all of the above methods, and allows for easy extensions to such a code. Extendibility, particularly in such a way as to incorporate additional physics, is important for the long-term evolution of software; if it cannot be updated to make use of advances in algorithms, underlying theory or increases in computational resources, it is unlikely to gain widespread use.

The case for ray-traced radiative transfer is strong, and indeed many implementations exist. Surprisingly, however, there is something of a dearth of codes developed specifically for SPH. SPHRAY is a notable example, primarily because it is publicly available[9] Some special attention is therefore given to it below, followed by a brief discussion of issues identified in its successor, P-SPHRAY.

SPHRAY

The code SPHRAY (Altay et al., 2008) is essentially an SPH-based implementation of the grid code CRASH2 (Maselli et al., 2009). It is a Monte Carlo ray tracing code, written largely in Fortran 90. SPHRAY is a full radiative transfer package, requiring as input a GADGET-2-like snapshot and configuration script(s), and the code is entirely *serial*. Each photon packet is *monochromatic*, but it can randomly sample frequencies from a provided spectral energy distribution in order to model non-monochromatic sources; similarly, only rays from a single source may be traced at any one time, but sources are randomly sampled weighted by their luminosity. The latter is achieved via inverse transform sampling from the luminosity cumulative distribution function of all sources.

SPHRAY is novel in that it accelerates its ray tracing with a hierarchical tree structure, specifically an *octree*, a technique borrowed from the field of computer graphics (CG)

[9] Originally at `https://code.google.com/archive/p/sphray/`, now archived; the author has a non-archival copy at `https://github.com/galtay/sphray`, and I maintain a modified fork at `https://github.com/spthm/sphray`.

(see Chapter 3). Further, it implements the fast ray-box intersection tests of Mahovsky and Wyvill (2004) for additional performance when *traversing* the octree.

One key issue that arises when tracing directly through an SPH field is how to calculate the optical depth. Below is described the method developed by Altay et al. (2008) for SPHRAY (and later used by Forgan and Rice (2010) and Altay and Theuns (2013), for example).

First, consider the column density along a ray from a source to some point of interest a distance L away. The density at any location in the SPH field may be approximated as Eq. (2.30), and so the column density is

$$N_{\rm cd} = \int_0^L \rho(\mathbf{r})\ \mathrm{d}l \approx \int_0^L \sum_{i=1}^{N} m_i\, W(r_{li},\, h_i)\ \mathrm{d}l, \tag{2.43}$$

where, as in Section 3.3, $r_{li} \equiv |\mathbf{r}_l - \mathbf{r}_i|$ with $\mathbf{r}_l \equiv \mathbf{r}(l)$ being the position a distance l along the ray, and the sum is over all N particles (with masses m_i) which are intersected by the ray. We then swap the order of summation and integration to give,

$$N_{\rm cd} \approx \sum_{i=1}^{N} \int_0^L m_i\, W(r_{li},\, h_i)\ \mathrm{d}l = \sum_{i=1}^{N} \int_{l_i^{\rm in}}^{l_i^{\rm out}} m_i\, W(r_{li},\, h_i)\ \mathrm{d}l, \tag{2.44}$$

where $l_i^{\rm in}$ and $l_i^{\rm out}$ are, respectively, the entry and exit distances along the ray for particle i. This last step is a further approximation, but should only lead to errors on the order of the smoothing length, which is of course the spatial resolution limit of the data itself. One can now see that the column density is a sum of line integrals through the SPH kernels of all intersected particles. Note that r_{li} is merely the impact parameter (distance of closest approach), b_i, of the particle for the ray.

The above method is utilized in both GRACE and, by extension, TARANIS, but is not the only option. For example, Kessel-Deynet and Burkert (2000) developed a different scheme. They decompose a ray into distinct points i, and compute the local *number* density at each such point via Eq. (2.30),

$$n_i = \sum_{j=1}^{N} n_j\, W(r_{ij},\, h_j), \tag{2.45}$$

where n_j is the number density of the photoabsorbing species of interest for particle j. These density values are then integrated by summation, somewhat similarly to Eq. (2.44).

However, in their scheme, optical depths rather than column densities are desired (recall Eqs (2.6) and (2.12)). The optical depth integral at each point k is estimated via the trapezoidal rule — though of course any method of approximating a definite integral is applicable — as,

$$\tau_k \approx \frac{1}{2}\bar{\sigma}(l_{k+1} - l_k)(n_{k+1} + n_k), \tag{2.46}$$

where n_k is as in Eq. (2.45) and l_k is the distance of point k along the ray. $\bar{\sigma}$ is the effective ionization cross-section of the photoabsorbing species, which is the true cross-section weighted by the spectrum of the source, S_ν, as

$$\bar{\sigma} = \frac{\int S_\nu \, \sigma_\nu \, \mathrm{d}\nu}{\int S_\nu \, \mathrm{d}\nu}. \tag{2.47}$$

The total optical depth between the source and a given point m is then simply $\sum_{k<m} \tau_k$.

While it is a viable candidate, I decided not to pursue a modification of the SPHRAY code in developing GRACE. For one, SPHRAY is written in Fortran, but there exists no freely-available Fortran compiler for code written for GPUs (see Chapter 3, Section 4). In any case, Fortran-C/C++ interoperability would likely be required, since GRACE uses C++ features not available in any version of Fortran. Finally, the ultimate goal with TARANIS is a fully-parallel ray-traced radiative transfer code, but the radiative transfer mechanism in SPHRAY is fundamentally serial in operation and would thus also need replacing.

P-SPHRAY

P-SPHRAY[10] (parallel-SPHRAY, developed primarily by Yu Feng) was suggested by Gabriel Altay (primary author of SPHRAY) as an alternative. It is an OpenMP-[11] (shared-memory)-parallelized rewrite of SPHRAY, written in C, making it an attractive option for solving the issues raised at the end of Section 4.3. Both ray tracing and the radiative transfer components are parallelized, but retain the same basic algorithms as SPHRAY. For ray tracing this is a fairly trivial operation, essentially prepending the serial code with a `#pragma omp parallel`. To overcome the radiative transfer algorithm's inherently serial nature, actual particle updates are protected by `#pragma omp atomic` declarations, which ensure that only one particle may be updated at a time, even if

[10] Available at `https://github.com/rainwoodman/psphray`, though no longer maintained.
[11] An application programming interface (API) for shared-memory *parallel* processing

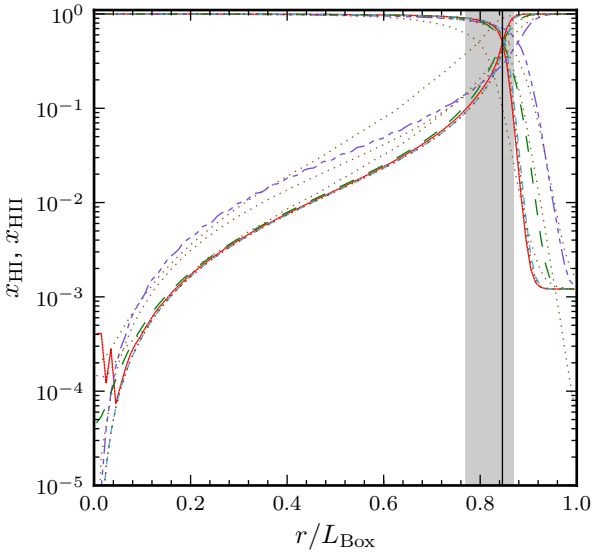


Figure 2.1. CRTCP Test 1. x_{HI} and x_{HII} radial profiles at $t = 500\,\mathrm{Ma}$. 1024 rays are traced per tick, and the total time of 500 Ma is divided evenly across 10^4 ticks. Results from P-SPHRAY are in red, CRASH in dash-dotted purple, RSPH in dashed green, and C^2-RAY is in dashed light-blue; all other codes which completed the test are in dark orange. The vertical black line denotes the $x_{\mathrm{HI}} = x_{\mathrm{HII}} = 0.5$ mark, and the shaded region covers $0.1 \leq x_{\mathrm{HII}} \leq 0.9$.

multiple *threads* are currently operating on that particle. However, according to Yu Feng (private communication, 2012-12-06), this does not scale well beyond approximately 30 CPU cores; that is, the SPHRAY algorithm is fundamentally not well-suited to parallel computation.

In order to assess its viability, P-SPHRAY was used to process a set of standard tests from the Cosmological Radiative Transfer Comparison Project (CRTCP) (see Chapter 5, Section 5 for a full description of these tests, which are only outlined here for brevity). The first test concerns the growth of an ionized hydrogen region in an initially-neutral hydrogen region, with a single monochromatic source emitting $\dot{N}_\gamma = 5 \times 10^{48}\,\mathrm{s}^{-1}$ ionizing photons at frequency $\nu = \nu_0 = 13.6\,\mathrm{eV}$, the ionization threshold of hydrogen. The gas is considered to be isothermal, held at $T = 100\,\mathrm{K}$. The ionization state at the end of this test can be seen in Figure 2.1. P-SPHRAY passes this test, with results closely matching those of other codes, and SPHRAY itself (which is not plotted here).

The second test is similar, and the initial conditions are identical, but the isothermal restriction is dropped, and the source is instead a $T = 10^5\,\mathrm{K}$ black body emitting 5×10^{48} ionizing photons per second. Both the ionization and the temperature state at the end of this test can be seen in Figure 2.2, where it is clear that P-SPHRAY's

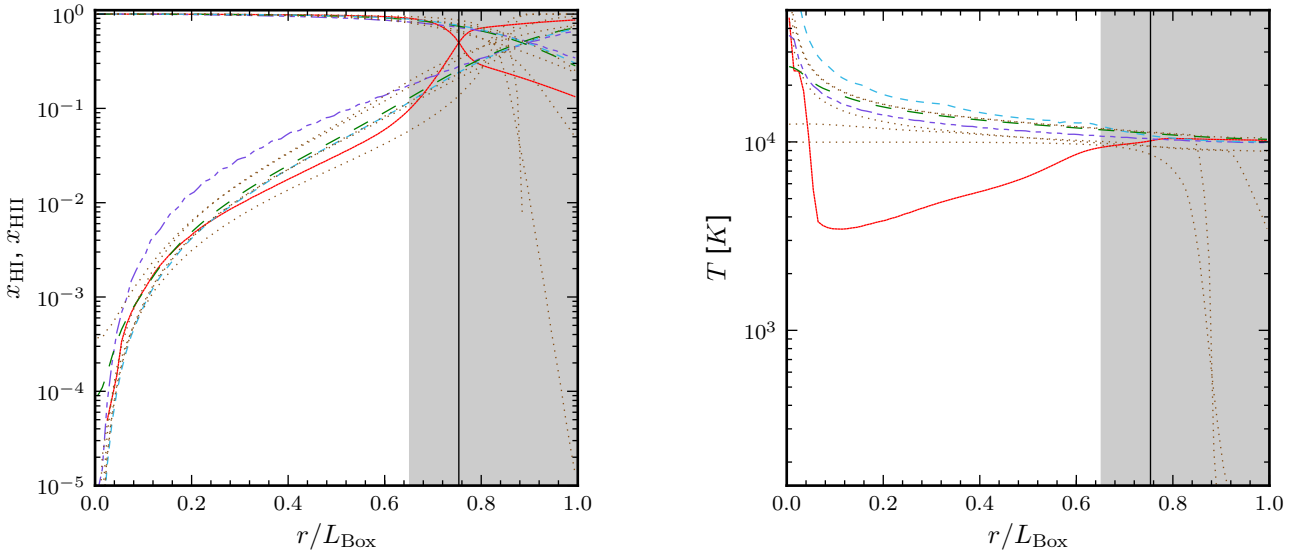
ionization front lags slightly behind that of the other codes, and more significantly, that it produces significantly lower temperatures within the HII region.

While the exact source of the problem was not determined, it was decided that the temperature solver in P-SPHRAY should be regarded as faulty. This alone may not have been sufficient cause to rule out the code completely, but a number of other (possibly related) issues were discovered. The `psphray` binary could not be compiled with OpenMP support, i.e. with support for parallel processing, as this consistently led to the code generating runtime errors and `nan` values in its output. Further, the optional LSODA[12] integrator could not be used as this also resulted in runtime errors. Only the built-in Euler integrator, in serial mode, was able to complete the test.

Finally, it is telling that P-SPHRAY support appears to have been dropped in favour of a P-SPHRAY version 2,[13] but that this is also in an unmaintained and incomplete state.

[12] One of several ordinary differential equation solvers developed by the Lawrence Livermore National Laboratory, now part of ODEPACK, which is available at `https://computation.llnl.gov/casc/odepack/`. P-SPHRAY uses (a modified version of) a C-conversion of LSODA due to Heng Li; this C version is available at both `https://github.com/lh3/misc/blob/master/math/lsoda.c` and `http://lh3lh3.users.sourceforge.net/download/lsoda.c`

[13] Available at `https://github.com/rainwoodman/psphray2`



(a) Radial profile of ionization state.

(b) Radial profile of temperature state.

Figure 2.2. CRTCP Test 2. $x_{\rm HI}$, $x_{\rm HII}$ and T radial profiles at $t = 500\,\mathrm{Ma}$. 1024 rays are traced per tick, and the total time of 500 Ma is divided evenly across 10^4 ticks. Results from PSPHRAY are in red, CRASH in dash-dotted purple, RSPH in dashed green, and C^2-RAY is in dashed light-blue; all other codes which completed the test are in dark orange. The vertical black line denotes the $x_{\rm HI} = x_{\rm HII} = 0.5$ mark, and the shaded region covers $0.1 \leq x_{\rm HII} \leq 0.9$.

3

Ray Tracing

1 Introduction

Ray tracing is, generally, a method of simulating the propagation of photons. In astrophysics, this includes ionization of hydrogen and helium (e.g. Cantalupo and Porciani, 2011), the transport of Lyman-α (e.g. Smith et al., 2015), dust radiative transfer (e.g. Steinacker et al., 2013), design of optical systems (e.g. Okumura et al., 2011), and computation of gravitational lensing statistics (e.g. Hilbert et al., 2009; Killedar et al., 2012).

In computer graphics (CG), ray tracing was originally developed in order to accurately visualize solid objects, giving an increased sense of depth relative to wireframe representations (Appel, 1968). Early implementations, now often categorized as *ray casting*, follow a ray from the camera (image plane) and stop as soon as the ray intersects an object. This basic method was extended to include more realistic illumination effects, producing new rays at the intersections to account for reflection, refraction and shadowing (Whitted, 1980). The recursive nature of such 'Whitted'-style ray tracing remains the essence of today's advanced ray-traced *renderings*. (Where rendering in this context means production of a 2D image from — usually — 3D models of objects, collectively a *scene*.) Multi-sampling, or *distribution ray tracing* (Cook et al., 1984), allows for many other effects, such as soft shadows, diffuse reflections, depth of field and motion blur. This is achieved via over-sampling rays in both the space and time domains, and is essentially a Monte Carlo method. It is here that ray tracing's advantages for image

synthesis are most apparent (Djeu et al., 2011). Photo-realistic images are typically produced through *physically based rendering* (see Pharr and Humphreys, 2010, ch. 1, for an introduction to this topic), which simulates the physical interaction of light and materials, and is thus an obvious application for ray tracing.

Rendering was formalized by Kajiya (1986) with the *rendering equation*, an integral equation which relates the intensity of light reaching and leaving a given point on a surface in a given direction. It is interesting to note that this is essentially the same as the equation of radiative transfer (Peters, 2014), Eq. (2.17).

The term ray tracing is, then, somewhat broad in definition. Here and throughout, it refers not to the general, recursive algorithm, but to the core tracing function itself. That is, the algorithm(s) which, given a set of spatial data and a set of rays, find intersections between the rays and objects in the data. This spatial data could be, for example, an adaptive mesh refinement (AMR) grid, smoothed particle hydrodynamics (SPH) field, unstructured mesh, or set of triangles — *primitives* in computer graphics parlance. *Rays* are simply lines with an origin, direction, and a length, or endpoint.

In Section 2 the basic concepts of ray tracing are covered, as it has been defined above. Relevant work in computer graphics is discussed in Section 3. Section 3.4 covers ray tracing methods in the astrophysics literature. In Section 4 an introduction to general-purpose computing on graphics processing units (GPGPU) is provided, followed by discussion of the ray tracing literature which is specific to GPUs.

2 Ray Tracing Fundamentals

The most obvious algorithm for finding intersections between a set of rays and a set of primitives is direct comparison. For N rays and M primitives, this solution scales as $O(NM)$, which rapdily becomes intractable. Before covering faster algorithms in Section 3, some common ray-primitive intersections are considered.

2.1 Ray and axis-aligned box intersection

As noted in Section 1, a ray may be defined by an origin and a direction. Of importance is the point(s) at which a ray intersects an object. It is typical to parameterize the ray (Pharr and Humphreys, 2010, ch. 1) as

$$\mathbf{r}(t) = \mathbf{O} + t\mathbf{d}, \tag{3.1}$$

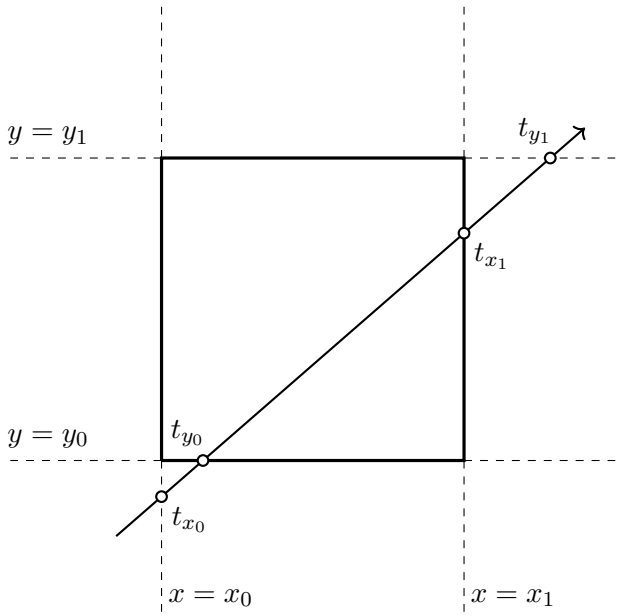


Figure 3.1. An illustration of the terms involved in the ray-box intersection test. The box is bounded by the lines $x = x_0$, $x = x_1$, $y = y_0$ and $y = y_1$. t_{x_n} and t_{y_n} denote the distance along the ray at which the ray intersects the $x = x_n$ and $y = y_n$ lines, respectively..

where **O** is the ray's origin, **d** is its *normalized* direction, and t is a parameter describing distance along the ray, often taken to lie in the range $[0, \infty)$. Note that Euclidean geometry is assumed throughout, but that ray tracing schemes in non-Euclidean spaces are possible (e.g. Weiskopf, 2000). Hence, when computing an intersection, one is computing the value of t at which a ray intersects an object. A simple — and highly relevant, see Section 3 — case is that of a ray intersecting an axis-aligned cuboid, or axis-aligned bounding box (AABB). That is, a cuboid whose six faces are all plane-parallel to the xy-, xz- and yz-planes of the co-ordinate system. A simple method for computing this intersection test is illustrated in Figure 3.1 and described below.

The bounding box is described by its 'top' and 'bottom' corners; in n-dimensions this requires $2n$ values to represent the minimum and maximum extent of the box along each co-ordinate axis. In 2D, we have $b_{\min} = (x_0, y_0)$ and $b_{\max} = (x_1, y_1)$. In order to determine ray intersection, we test the ray against each of the four lines which bound the box. Comparing to the lines of constant-y in Figure 3.1, we have,

$$r_y(t) = O_y + t_{y_0}\, d_y = y_0, \tag{3.2a}$$

$$r_y(t) = O_y + t_{y_1}\, d_y = y_1, \tag{3.2b}$$

where r_k, O_k and d_k refer to the kth co-ordinate of $\mathbf{r}$, $\mathbf{O}$ and $\mathbf{d}$ as defined in Eq. (3.1), and t_{y0} and t_{y1} are the values of t at which the ray intersects the bottom and top y-boundaries, respectively. It should be noted that we cannot assume $t_{y_0} < t_{y_1}$, which clearly does not hold for a ray originating from the upper-right of Figure 3.1 with direction $-\mathbf{d}$. The solutions to these equations, and the analogous equations for bounding lines of constant-x, are

$$t_{x_0} = (x_0 - O_x) \,/\, d_x, \tag{3.3a}$$

$$t_{x_1} = (x_1 - O_x) \,/\, d_x, \tag{3.3b}$$

$$t_{y_0} = (y_0 - O_y) \,/\, d_y, \tag{3.3c}$$

$$t_{y_1} = (y_1 - O_y) \,/\, d_y. \tag{3.3d}$$

Finally, we compute the box-entry (t_{min}) and box-exit (t_{max}) distances along the ray as

$$t_{\mathrm{min}} = \max(t_{x_0},\ t_{y_0}), \tag{3.4a}$$

$$t_{\mathrm{max}} = \min(t_{x_1},\ t_{y_1}). \tag{3.4b}$$

We then have an intersection if and only if

$$t_{\mathrm{min}} < t_{\mathrm{max}}. \tag{3.5}$$

In three dimensions, Eqs (3.2) and (3.3) are all extended to include similar relations for the z-component, and we have,

$$t_{\mathrm{min}} = \max(t_{x_0},\ t_{y_0},\ t_{z_0}), \tag{3.6a}$$

$$t_{\mathrm{max}} = \min(t_{x_1},\ t_{y_1},\ t_{z_1}), \tag{3.6b}$$

before determining intersection via Eq. (3.5).

There are substantially more advanced algorithms available, for example that of Mahovsky and Wyvill (2004), which relies on Plücker co-ordinates and is used in SPHRAY (Altay et al., 2008). The ray-slopes method of Eisemann et al. (2007) claims to be yet faster still, and was used by Forgan and Rice (2010) (though in practice I have found this to not be the case, see Chapter 4, Section 7.3, page 130). However, these advanced methods generally only show an advantage on the CPU, since they involve *branching*

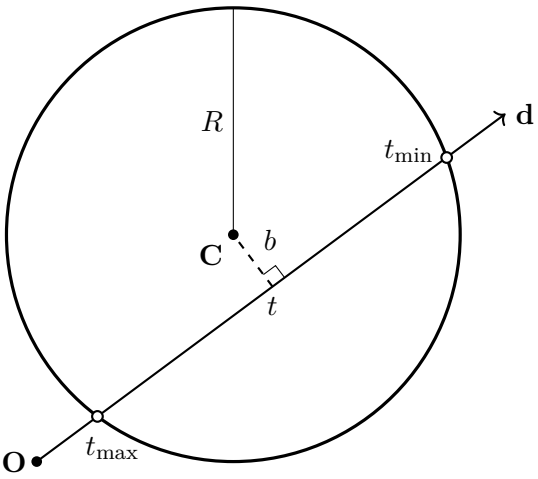


Figure 3.2. An illustration of the terms involved in the ray-sphere intersection test. The sphere is of radius R, with centre $\mathbf{C}$. $t_{\min}$ and $t_{\max}$ denote the distance along the ray at which the ray enters and exits the sphere, respectively. t denotes the distance along the ray to the point of closest approach, and b the impact parameter. Here a slice through the sphere is shown, such that the ray's direction vector $\mathbf{d}$ lies entirely within the slicing plane, and the plane contains the point $\mathbf{C}$.

code. As will be shown in Chapter 4, Section 7.3, branchless variants of the simple method presented above perform best on the GPU.

2.2 Ray-sphere intersection

Intersecting rays with spheres is another simple example, described here because of its application to SPH. Using again the parameterization in Eq. (3.1), a ray intersects a sphere of radius R centred on the origin where

$$\begin{aligned} r_x(t)^2 + r_y(t)^2 + r_z(t)^2 - R^2 &= 0 \\ (\mathbf{O} + t\mathbf{d})_x^2 + (\mathbf{O} + t\mathbf{d})_y^2 + (\mathbf{O} + t\mathbf{d})_z^2 - R^2 &= 0 \end{aligned} \tag{3.7}$$

is satisfied, as illustrated in Figure 3.2. This is a quadratic equation in t, where the smallest real root is the entry point, $t_{\min}$, and the largest real root is the exit point, $t_{\max}$. If there are no real roots, the ray misses the sphere (Pharr and Humphreys, 2010, ch. 1).

In some cases, and in particular for GRACE, it may be more useful to know the t-distance to the point of closest approach of the sphere's centre. That is, the value of t for which $\mathbf{r}(t) - \mathbf{C}$ is minimized, where $\mathbf{C}$ is the sphere's centre. This can be efficiently computed via a dot product as

$$t = (\mathbf{C} - \mathbf{O}) \cdot \mathbf{d}, \tag{3.8}$$

and we can then compute the *impact parameter* (the minimum value of $\mathbf{r}(t) - \mathbf{C}$) as

$$b = |(\mathbf{C} - \mathbf{O}) - t\,\mathbf{d}|. \tag{3.9}$$

The final tests for intersection are then,

$$0 \leq t \leq l, \tag{3.10a}$$

$$b \leq R, \tag{3.10b}$$

where l is the length of the ray. Both relations must hold for an intersection. Note also that $<$ and $\leq$ are somewhat interchangeable here, depending on one's precise definition of 'intersection'.

3 Ray Tracing Techniques from Computer Graphics

Due to its ability to create highly-realistic images and high computational cost (Bikker, 2007), much effort has been invested by the computer graphics community in developing both more accurate and more computationally efficient ray tracing schemes. The latter of these two efforts saw renewed interest in the early 2000s (Lauterbach et al., 2006), due to the proliferation of computing hardware sufficiently powerful to perform interactive ray tracing (of simple scenes) (e.g. Reshetov et al., 2005; Wald et al., 2007a). A similar increase in research effort for ray tracing on GPUs began in the late 2000s, in part due to the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) framework (Nickolls et al., 2008), vastly simplifying the programming of GPUs for general tasks relative to earlier efforts, (e.g. Carr et al., 2002, 2006; Foley and Sugerman, 2005; Purcell et al., 2002).

Following the lead of SPHRAY (Altay et al., 2008), whose ray tracing component is influenced by techniques in computer graphics, I have drawn on much of the more recent work in development of GRACE, and in this section cover the most relevant aspects.

3.1 Acceleration Structures

Probably the most significant contribution to the high performance of modern ray tracers is that of an *acceleration structure*. All such methods aim to reduce the number of ray-object intersection tests that are performed (Amanatides and Woo, 1987) and all fall into two categories (Wald et al., 2007b): *spatial partitioning* and *object partitioning*.

Whichever approach is chosen, each partition has some volume associated with it, and in the case that this volume is a 'simple' shape, it can be intersected with the ray relatively efficiently (e.g. see Section 2.1) — typically more efficiently than the objects within it. When a ray misses such a volume it follows that the ray must also miss all elements contained within it, thus removing, potentially, many primitives from the search. All acceleration structures attain their performance advantages in this manner, though there are many variations. The three most common approaches are outlined below, but see, for example, Pharr and Humphreys (2010, ch. 4) for a more detailed discussion and implementation notes; Szirmay-Kalos et al. (2002) for an analytic investigation of the efficiency of spatial partitioning schemes; Wald et al. (2007b) for a description of the various costs and benefits of space vs. object partitions; and references contained below.

Typically, acceleration structures are hierarchical — a *tree*-like structure of *nodes*. At the lowest (finest) level are *leaf nodes*. A leaf contains some number (≥ 1) of the objects over which the acceleration structure is built, according to the splitting method used. Each leaf node has a *parent node*, which is also an *inner node*; all non-leaf nodes are inner nodes. This inner node has its own parent node, and so on until the root node, which is unique both because it has no parent node, and because it is the only common *ancestor* of all leaf nodes. The number of *child nodes* for which an inner node is a parent is usually fixed for a tree, and known as its *degree* or *arity*. Common examples are *binary trees* (two child nodes per inner node), *quadtrees* (four, convenient for spatial partitioning of two-dimensional data) and *octrees* (eight, convenient for spatial partitioning of three-dimensional data).

Regular grids are the simplest acceleration structure, and atypical in that they are not hierarchical. They generally provide the lowest performance, suffering when the geometry is not uniformly distributed (e.g. Günther, 2014, pp. 24). Regular grid subdivision is a spatial partitioning scheme. The domain is divided into equal-sized boxes, also known as cells or *voxels*, with each voxel having references to all primitives

which overlap it. (That is, a given element may be referenced by multiple voxels.) Rays then need only be intersected against the primitives that are referenced by intersected cells.

This method is typically combined with a *ray marching* technique to visit all the cells in-order with respect to the ray's origin and direction. The ray's entry point to the entire domain gives the location of the starting voxel. The exit location of this voxel then defines the entry point of the next voxel, and so on. In fact, it is possible to traverse from one voxel to the next using only two floating point comparisons and one floating point addition (Amanatides and Woo, 1987). The process is similar to computation of the various t_{x_n}, t_{y_n} and t_{z_n} values, as described in Section 2.1. In 2D, we compute the t-distances at which the ray crosses the closest x and y boundaries; whichever gives the smallest t-value is the direction in which to move. For example, suppose our ray is in the bottom-left cell in Figure 3.1; we first cross at the point labelled t_{x_0}, and hence the t_x value is smaller and we must move to the neighbouring voxel in the x-direction. Indeed, the cell to the right of the current cell is the next-intersected cell.

Introducing a z-dimension simply adds an extra comparison operation. In graphics applications, the *closest* intersection of a ray with an object is generally all that is required (Cleary and Wyvill, 1988), and traversing the grid in-order — with respect to the ray's origin and direction — then clearly has a significant advantage: as soon as an object is intersected, the search can end. As will be discussed in Section 3.4, this is not the case for many astrophysical ray-tracing codes.

A general, and notable, disadvantage of the regular grid is its multiple referencing of primitives, which will often result in the same ray being tested for intersection against the same primitive several times. Amanatides and Woo (1987) presented a method for eliminating multiple intersection tests of the same element: each primitive records the ID of the most recent ray to have performed an intersection test against it. Unfortunately, this is clearly not suitable for a *massively parallel* environment in which multiple rays are traced simultaneously. Additionally, this algorithm is also not suitable for a parallel single-instruction multiple-data (SIMD) implementation, which is necessary to attain maximum performance on modern CPU cores. The performance of more advanced grid-*traversal* algorithms, which are parallel in nature, is of the same order of magnitude as k-d tree traversal (see below), though still somewhat slower (see Wald et al., 2006, for example).

Regular grids can offer an overall advantage in rendering dynamic scenes, as they can be constructed quite efficiently, offsetting their reduced tracing performance (again, see Wald et al., 2006); other approaches refrain from rebuilding entirely, instead updating the grid as needed (Reinhard et al., 2000). For the same reason, grids were also an early target for on-GPU construction of acceleration structures (e.g. Kalojanov and Slusallek, 2009).

Uniform grids can be traced with $O(N^{1/3})$ time complexity, for N primitives, depending on the geometry (Ize, 2009, ch. 4).

***k*-d trees** (Bentley, 1975; Havran, 2001, ch. 4) are a restricted form of binary space partitioning (BSP) tree, a hierarchical data structure in which the partitions are axis-aligned; due to the resulting simplicity in searching the tree, essentially all BSP trees used in ray tracing are k-d trees (Ize, 2009, ch. 2).

In the traditional k-d tree, the dataset is recursively subdivided in space, with the partitioning cycling through the available dimensions; in ray tracing, these are the spatial x-, y- and z-dimensions. In practice, the cyclic requirement may be omitted in order to better balance the resulting tree (for instance, see Fussell and Subramanian, 1988). k-d trees are thus able to adapt to highly non-uniform datasets, but, similarly to the uniform grid, a single element may overlap — and therefore be referenced by — multiple cells of the tree.

The criteria for terminating the subdivision are specific to the implementation, as are the locations of the splitting planes. Both of these choices can have a significant impact on the ray tracing performance (Wald and Havran, 2006). For the latter, typically the surface area heuristic (SAH) (MacDonald and Booth, 1990) is employed, which attempts to minimize the cost of traversing the tree as a whole (Havran and Bittner, 2002) (see Section 3.2). The traversal itself is relatively simple; it is sufficient to test the ray against only the splitting plane of a cell to find which partitions a ray intersects, rather than testing against both partitions' AABBs (Fussell and Subramanian, 1988). As a single AABB test is effectively six ray-plane tests, this represents a substantial reduction in computation.

Finally, much like for the uniform grid, k-d trees offer an advantage when only the closest intersection along the ray is required (Wald et al., 2007b). They can easily be traversed in-order, exiting the search as soon as an intersection is found.

For a nearest-neighbour search, k-d trees covering N items exhibit typical running times of $O(\log N)$ (Bentley, 1975).

Bounding volume hierarchies (BVHs) partition objects rather than space. The object space — the set of all primitives in the dataset through which we trace — is recursively subdivided into nodes, forming a tree of nodes. Every node has an associated volume, which tightly bounds the elements it contains, and is generally simple to test for intersection against a ray, e.g. an AABB (Goldsmith and Salmon, 1987). Unlike a k-d tree, these boxes may, and typically do, overlap in space. For the following discussion, it is assumed that this bounding box is indeed an AABB.

This partitioning in object space is often achieved by selecting a splitting plane in real space, similarly to k-d trees. Each primitive is then *uniquely* assigned to one of the partitions, most frequently by selecting the partition in which a primitive's *centroid* (its geometric centre) lies (Wald, 2007). The subdividing process terminates subject to some criterion, a simple example being when there are fewer primitives in a node than a given threshold; a leaf node is thus formed.

The number of primitives in each leaf can affect the ray tracing efficiency of the BVH structure: when ray-primitive intersection tests are more computationally expensive than ray-AABB tests, a deeper tree with fewer primitives per leaf will be more effective; if the ray-primitive test is relatively cheap, a shallower tree with larger leaves may prove most efficient (Lauterbach et al., 2006). One must also take hardware-specific considerations into account; see Chapter 4, Section 2 for further discussion as it applies to GRACE. In the 'classic' BVH, no object will be referenced by more than one leaf. However, some algorithms do split primitives across multiple leaves of the BVH; this will not be discussed here, but see, for example, Stich et al. (2009).

The selection of the splitting plane will also — as for k-d trees— have an effect on ray tracing performance. A simple method is to select the spatial median of all the primitives in a node along a particular axis. The SAH can also be applied to BVHs (first noted by Müller and Fellner (1999), though see e.g. Wald (2007) for a more recent implementation). In fact, minimizing an area-estimated cost during tree construction was proposed first for BVHs, by Goldsmith and Salmon (1987). Similarly to k-d trees, it usually results in the highest-performing BVHs when compared to other methods. For a comparison of the most common techniques for selecting primitive partitioning during BVH construction, see Wald et al. (2007a).

BVH traversal proceeds as follows: if a ray intersects the AABB of some node, A, then the ray is tested for intersection with the child nodes of A. The process is repeated for each intersected child node, and so on, until a leaf node is reached. If a leaf node is intersected, all primitives it contains are tested for intersection with the ray.

While k-d trees have been typically thought of as achieving the highest ray tracing performance (and note that Havran (2001), pp. 44, in one of the first rigorous investigations of the efficiencies of different acceleration structures, found that BVHs resulted in rather poor performance), some authors have presented BVH schemes which achieve similar performance to k-d trees (e.g. Wald et al., 2007a).

Traditionally, it has been in deformable or dynamic scenes, where the geometry and topology may be frequently altered, that BVHs show a distinct advantage (Lauterbach et al., 2006; Wald et al., 2007a, for example). This is, in part, because they can be updated after small changes in the positions of primitives without requiring a complete rebuild, a technique known as *refitting*. During a refitting, only the AABBs of nodes are updated; no changes to the partitioning or hierarchy are made. A more advanced approach is that of *partial restructuring*, in which small subsets of the tree are reconstructed, if required (Yoon et al., 2007). As will be discussed in Section 4.2, the most recent BVH construction algorithms on GPUs are able to produce high-quality BVHs in a short enough time that they can be rebuilt every frame (every 16.67 ms when displaying at 60 frames per second), even for relatively complex scenes.

The traversal of a BVH tree covering N primitives has $O(\log N)$ time complexity (Garanzha and Loop, 2010).

3.2 The Surface Area Heuristic

Beyond acceleration structures themselves, a significant amount of literature exists in attempting to produce an optimal hierarchy within a given acceleration structure. The surface area heuristic (SAH) is a particularly noteworthy example, and described here for BVHs.

Originally proposed by Goldsmith and Salmon (1987), the method aims to minimize the expected cost of traversing a tree with a random ray. In essense, they noted that the probability of intersecting any given node is proportional to the ratio of that node's surface area to the total available surface area (i.e., the surface area of the root node). The total *SAH cost* of a tree is then estimated as (Wald et al., 2007a)

$$C = \sum_n 2\frac{A(n)}{A(\text{root})} C_{\text{AABB}} + \sum_l \frac{A(l)}{A(\text{root})} N(l)\, C_{\text{primitive}}, \tag{3.11}$$

where n denotes an inner node, l a leaf node, $A(x)$ is the surface area of node x, $N(x)$ is the number of primitives within node x, C_{AABB} is the cost of intersecting a single AABB, and $C_{\text{primitive}}$ is the cost of intersecting a single primitive. These last two values can be intuitively expressed as runtimes, though typically one is set equal to one, and the other set such that the ratio $C_{\text{AABB}}/C_{\text{primitive}}$ is equal to the ratio of runtimes.

Globally-optimizing this expression for a tree of any useful size is intractable, and it is often solved via top-down, greedy optimization. Construction begins at the root node, and whenever a node is considered for splitting into child nodes, we attempt to minimize the *local* SAH cost. This is obtained by splitting the current node, n, into two leaf nodes, l_L and l_R, such that Eq. (3.11) is minimized for the resulting two-leaf tree. That is, we attempt to minimize (Wald et al., 2007a)

$$C_{\text{local}}(n) = 2\, C_{\text{AABB}} + \frac{A(l_L)}{A(n)} N(l_L)\, C_{\text{primitive}} + \frac{A(l_R)}{A(n)} N(l_R)\, C_{\text{primitive}}, \tag{3.12}$$

which is simply the SAH cost for a two-leaf tree with root node n. This task is itself far from trivial when $N(n)$ is large. Typically, one uses heuristics to search for a good, rather than optimal, partitioning (e.g. Wald, 2007). Finally, when the cost of Eq. (3.12) exceeds $N(n)\, C_{\text{primitive}}$, we decide that n should become a leaf node.

3.3 The degree of acceleration structures

The previous sections discussed recursive partitioning of (object-)space, and primarily assumed two partitions per node; this is not a fundamental constraint, but specified by the *degree* of the tree. Put simply, it is the maximum number of child nodes any node may have; for example, an octree is of degree eight.

Repeated here is an analytical argument of Klosowski et al. (1998) advocating the use of degree-2 trees. First, assume a perfectly balanced tree of degree α, in which N primitives are distributed evenly across all leaf nodes, and all nodes have exactly α child nodes. During traversal, we have $\log_\alpha N$ levels and α nodes per level. The computational time required to traverse from the root node to a leaf is then

$$T \propto \alpha \log_\alpha N. \tag{3.13}$$

Expressed as a natural logarithm,

$$T \propto \alpha \frac{1}{\ln \alpha}, \tag{3.14}$$

which is minimized for $\alpha = e$, provided $\alpha \in (1, \infty)$. In reality we obviously require $\alpha \in [2, \infty) \cap \mathbb{N}$, finding $\alpha = 3$ minimizes the work. The choices of $\alpha = 2$ and $\alpha = 4$ predict equal time complexity in this simple model, slightly greater than that of $\alpha = 3$, but have the advantage of being powers of two. Due to its simplicity and near-optimal performance, $\alpha = 2$ is thus the most attractive option, and is often referred to as a binary tree. For example, it allows for comparatively straightforward choices of splitting planes. This decision mirrors the current practices in computer graphics, and in particular, k-d trees have been shown to be more efficient than octrees in both that domain (Havran, 2001, ch. 3) and astrophysics (Saftly et al., 2014).

It is worth mentioning that octrees are commonly used in N-body codes in order to accelerate the computation of gravitational forces between particles (see Chapter 2, Section 3.1 and, for instance, Bagla (2005) and references therein). Such methods are often referred to as *tree codes* or *Barnes-Hut trees*, crediting the authors of the original paper (Barnes and Hut, 1986). The tree method — including tree construction — has even been implemented on the GPU (Bédorf et al., 2012). Gravitational attraction is of utmost importance in astrophysical simulations, and many hydrodynamics codes employ N-body solvers for this purpose; one might then wonder if we cannot simply use an existing octree to accelerate ray tracing.

As an example, SPHRAY (Altay et al., 2008) uses an octree for its acceleration structure, and they make a similar observation. In reality, this would require supplementing the tree with AABBs, and a fall-back for codes not utilizing a Barnes-Hut tree. It would also likely restrict us to immediate compatibility with only one hydrodynamics code. However, as will be shown in Section 4.2, it is possible to construct the desired binary trees on the GPU with negligible overhead.

3.4 Ray Tracing in Astrophysics

Ray tracing in astrophysical simulation codes typically employs somewhat different techniques and optimizations (if any) compared to those covered in Section 3, so they are discussed here.

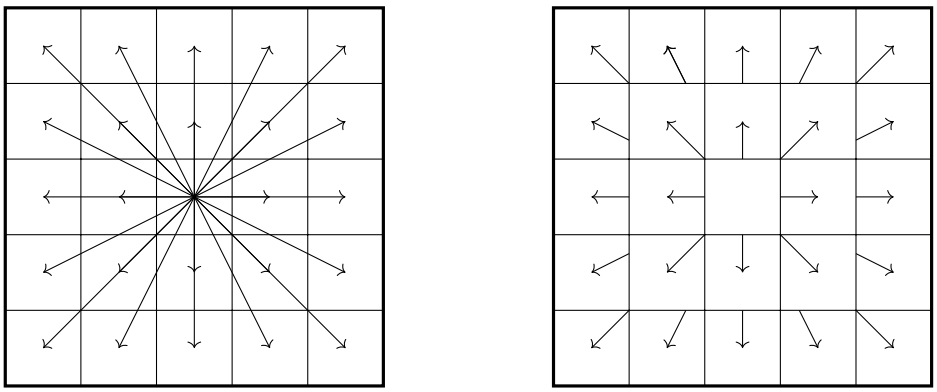


Figure 3.3. A 2D illustration of long characteristics (left) and short characteristics (right) ray tracing methods. In short characteristics, only one ray is traced through each cell. A ray's incoming flux is interpolated from the outgoing flux vectors of neighbouring cells..

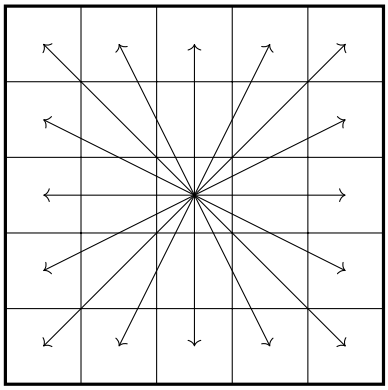


Figure 3.4. A 2D illustration of optimal long characteristics ray tracing. Rays are cast to only the most distant cells from the source, but affect all cells through which they pass.

In the simplest case we have the method of *long characteristics*, as shown on the left of Figure 3.3; in its most straightforward implementation, a ray is cast from each source to each simulation element, accumulating optical depth along its path. For every volume element, it provides an accurate optical depth to all sources, which is necessary for accurate radiative transfer. This is clearly inefficient, however, as elements close to a source will be intersected by many rays, wasting computation. A better solution is to only cast enough rays to sufficiently sample elements at large distances from a source (for N cells, this requires $O(N^{2/3})$ rays per source), but to process all cells along a ray's path rather than just its end point (see, for instance Abel et al., 1999), as is shown in Figure 3.4.

The method of *short characteristics* (Kunasz and Auer, 1988), shown on the right of Figure 3.3, aims to yet further reduce the workload, by tracing rays only across individual cells. To estimate the optical depth to a particular source, a cell interpolates

between the optical depths computed for the nearby cells which are upstream of it (i.e. in the direction of the source). It then traces a ray through itself to determine its contribution to the optical depth, which will be required by downstream neighbours. One will note that this necessitates some serial computation, as the input to each cell is a function of the output of cells closer to the source.

Tracing a ray through a cell can be reduced to only the ray-box intersection operations in Section 2.1 (computing $t_{\min}$ and $t_{\max}$ is sufficient to calculate the distance a ray travels through a cell), so in practice a short characteristics algorithm will not resemble a generic ray tracer. Mellema et al. (2006), for example, use the short characteristics method in C^2-RAY. As noted in Chapter 2, Section 4.3, they also use an implicit time-stepping scheme, which is efficient but not easily parallelizable, due to its causal nature. Hence neither the ray tracing nor the solver are particularly suitable for a massivley-parallel implementation.

Perhaps the most advanced technique is that of *adaptive ray tracing* (Abel and Wandelt, 2002). It produces a near-optimal use of long characteristics, providing even sampling over a solid angle without the interpolation errors of short characteristics. A tree of rays is formed, with rays splitting into sub-rays to ensure sufficient sampling of a cell — that is, as rays propagate farther from their source, they will be recursively split to ensure all cells are well-sampled. The sub-division of rays is done using the HEALPix code (Górski et al., 2005), which tiles equal-area 'pixels' over the surface of a sphere, up to arbitrarily high resolutions. The normal vector of a pixel gives the direction of a ray and, when a ray is split into (four) child rays, the corresponding sub-division of the pixel gives the new normal vectors.

Adaptive ray tracing has obvious advantages at the cost of increased implementation complexity. One subtlety is that, after splitting a ray, some of the new child rays may in fact intersect different objects than their parent. In addition, geometrical artefacts can occur due to the imperfect alignment of volume elements and HEALPix pixels; this requires either more rays or a geometric correction factor to compensate (Wise and Abel, 2011, and note that, here, the factor breaks photon conservation, though to negligible effect).

Finally, note that not much attention has been given to efficient execution of the ray tracing algorithm itself, as we do here. Two notable examples are Saftly et al. (2013), comparing various octree construction and traversal algorithms, and Saftly et al. (2014),

comparing octrees to k-d trees. They ultimately advocate k-d trees, combined with physically-based conditions for halting the subdivision during construction.

4 Graphics Processing Units

Due to its highly parallel nature, ray tracing is an obvious target for computation on graphics processing units (GPUs), highly-parallel processors with up to several thousands of cores on a single chip. Before introducing some GPU-specific ray tracing techniques, this section covers both the hardware and software aspects of GPUs, both of which are effectively encompassed by the Compute Unified Device Architecture (CUDA) programming paradigm (Nickolls et al., 2008). This provides context for the optimizations employed in GRACE (see Chapter 4).

It should be noted that OpenCL is a comparable alternative to CUDA programming. GRACE is written in CUDA C++ for NVIDIA GPUs, and so that model is described here, but many of the key lessons are applicable to both. As for the choice of CUDA itself, the aim is the maximum achievable performance, which naturally requires that one can take full advantage of the hardware. CUDA best meets this need, exposing hardware functions that would be difficult to incorporate into the more portable OpenCL, which also targets multicore CPUs and heterogeneous processors like the Cell Broadband Engine, among others (Stone et al., 2010).

General-purpose computing on graphics processing units (GPGPU) has now become somewhat commonplace in high performance computing (HPC), and many GPU codes have been developed within the field of astrophysics: N-body simulations (Bédorf et al., 2012, 2014; Grimm and Stadel, 2014), radiative transfer (Aubert and Teyssier, 2010; Heymann and Siebenmorgen, 2012), near-earth asteroid detection (Shao et al., 2014), computation of cosmological statistics (Bard et al., 2013), astrophysical object classification with genetic algorithms (Cavuoti et al., 2014) and visualization (Hassan et al., 2013; Rivi et al., 2014), to name several more recent examples.

4.1 CUDA and NVIDIA Hardware

GPU hardware is often referred to as *massively parallel* or *massively threaded*, and a single modern GPU chip may contain $> 10^3$ computing cores, or *stream processors*. These cores are grouped into *multiprocessors* (variously SMs, SMXs or SMMs, depending

on the architecture), along with memory access and special-function units; a typical NVIDIA GPU will have $O(10)$ multiprocessors with $O(100)$ cores per multiprocessor. It is typical in both GPU and CPU parallel environments to refer to the basic unit of processing as a *thread*. From the programmer's perspective, parallel program execution gets no finer-grained than the thread. If we launch a program on 10 threads, all 10 threads are executing the same program. In CUDA, it is typical to launch thousands of threads, and the program to be executed is known as a *kernel*. The kernel is written only once, yet it prescribes the behaviour of all threads. Each thread has a unique identifier, a *thread ID*, use of which primarily enables different threads to load different data.

Threads are themselves grouped into *warps*, consisting of 32 threads on all current NVIDIA hardware. The warp is particularly important, as all threads in a warp execute synchronously. That is, they are always at the same point within the kernel. If the kernel logic requires that threads within a warp execute different instructions — a behaviour known as *warp divergence* — all threads in the warp actually execute *all* instructions, but threads which were not intended to execute a particular instruction have their results (i.e. writes to variables or memory) discarded.

For example, suppose threads 0 – 15 wish to compute $c = a + b$, but threads 16 – 31 wish to compute $c = a - b$. All threads 0 – 31 first compute $a + b$, but only threads 0 – 15 save the result; likewise, all threads then compute $a - b$, but only threads 16 – 31 save the result. This doubles the computation time relative to what the programmer would naïvely expect and, importantly, it holds even if only one of the 32 threads is divergent. It is therefore very desirable to have threads in the same warp follow the same path through the program.

Similar best-practices extend to memory operations, as reads and writes to the main on-device memory perform best if a warp accesses a *contiguous* block of memory; this is known as *coalescing*. Note there is no requirement that consecutive threads access consecutive elements, only that the data requested by the warp as a whole be contiguous.

Warps are largely abstracted away from the programmer, but some warp-wide functions do exist. Fermi-class hardware[1] and newer contain warp-wide voting functions,[2]

[1] In chronological order, the relevant NVIDIA CUDA architectures are Fermi, Kepler, Maxwell, Pascal and Volta; Volta was released in 2017 and is not further discussed here.

[2] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-vote-functions`

which may be used to evaluate a *predicate* (a boolean value) over all threads in a warp. Kepler introduced warp shuffle functions,[3] which exchange 4-byte data (`int`, `float` etc.) between threads in a warp.

At the software level, the next — and most useful — level of granularity is known as a *block*, which is simply a collection of warps. At the hardware level, blocks are assigned to multiprocessors for execution. Each block can run on only one multiprocessor, but a single multiprocessor can process multiple blocks concurrently. The maximum number depends on the hardware architecture (currently 8, 16 or 32). Blocks are further grouped into *grids*, which are purely an abstraction (the hardware is unaware of them). The maximum size (number of threads) of a block and the maximum number of blocks a kernel can be launched with are architecture-specific. Similarly, the best-performing configuration of thread-block size and number of blocks tends to vary between architectures, even for the same kernel. General rules of thumb are thread blocks of at least 128 threads (i.e. 4 warps) per block and at least as many blocks as there are multiprocessors, ideally many more.

Blocks are significant as all threads in a block have access to the same *shared memory*, a small amount of fast, per-multiprocessor on-chip memory. A block may request up to 48 KiB of shared memory. On older architectures, this is the total size of shared memory available to the host multiprocessor. Some newer architectures have 64 KiB, 96 KiB or 112 KiB of on-chip memory; this only increases the number of blocks which may be concurrently running on a given multiprocessor, because blocks are still limited to 48 KiB of shared memory. Shared memory should be contrasted to the much slower *global memory*, which is typically several gigabytes in size.

The Fermi architecture introduced an *L1 cache*, which shared its physical on-chip storage with shared memory; the L1 and shared memory can be split as 48 KiB / 16 KiB, or vice-versa. Kepler added the option of a 32 KiB / 32 KiB split; however, loads from global memory (on-GPU RAM) are not cached in L1 on Kepler, with L1 being reserved for *register spills*, temporary storage when not all in-flight data will fit in registers. (Some Kepler chips can be switched into Fermi-like global caching in L1 with a compile-time flag.)[4] Maxwell has on-chip storage fully dedicated to shared memory,

[3] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-shuffle-functions`

[4] For Kepler-specific details, see the Kepler tuning guide:
`http://docs.nvidia.com/cuda/kepler-tuning-guide/`

and the L1 cache and *texture cache* have been merged into the same physical unit.[5] Pascal also has this unified cache.[6]

Shared memory is a way for threads within the same block to communicate and work co-operatively. Unlike within a warp, threads within the same block are not guaranteed to execute concurrently, and in practice they do not. Block-wide synchronization calls are available, forcing threads in a block to wait until all threads have reached that point.

In addition to being substantially faster, shared memory has different access characteristics to global memory. Consecutive 4-byte data elements are assigned to consecutive *banks*, with current hardware having 32 4- or 8-byte banks. Supposing a 4-byte element at index $i = 0$ is in bank 0, then element with index j is in bank $j \bmod 32$. A warp can access all 32 banks simultaneously, but accesses to the same bank are serialized; when multiple threads in a warp attempt to access the same bank, it is known as a *bank conflict*. Thus, in the ideal case, each thread within a warp reads from a different bank. It is also therefore possible to achieve maximum shared memory throughput without a warp accessing a contiguous block of data. In fact, for ease of indexing, it is typical to assign a contiguous block of shared memory to each thread. Bank conflicts can be easily avoided by ensuring that the number of elements in each thread's block of memory, k, has no common factors with the number of banks, 32. It is therefore sufficient to ensure $k = 2n + 1$ (i.e. odd, with padding in the case that only $k = 2n$ elements are actually required).

Another key aspect of GPU programming is *latency hiding*. A warp will typically have to wait many clock cycles between making a request for data in memory and that data being received (memory-access *latency*); if the next set of instructions depend on that data, the warp will *stall*; if no other warps are ready to execute instructions, then the multiprocessor will stall. However, in general, only a few warps can have instructions issued in a given clock cycle (exact numbers are architecture specific). Thus, if a sufficient number of warps always have instructions they can execute, any stalling of other warps can be completely hidden. This is latency hiding. Latency hiding can be most easily achieved by having large thread blocks, which increases the number of warps running on each multiprocessor. The utilization (how many warps are currently doing

[5] For Maxwell-specific details, see the Maxwell tuning guide:
`https://docs.nvidia.com/cuda/maxwell-tuning-guide/`

[6] For Pascal-specific details, see the Pascal tuning guide:
`https://docs.nvidia.com/cuda/pascal-tuning-guide/`

work relative to the maximum possible given hardware limitations) of the multiprocessor is known as the *occupancy*, and in general higher occupancy is better. In practice, an occupancy of $\sim 60\%$ is often sufficient to saturate the available memory bandwidth.

A typical CUDA program will read the relevant data from global memory into shared memory, coalescing global reads as much as possible, process the data, and finally write any results back to global memory, again coalescing where possible. Using shared memory as working memory reduces stalls, and can thus increase multiprocessor utilization and occupancy.

As a final note on GPU hardware — and to highlight the difficulty in synchronizing threads across the entire device — note that because L1 cache is per-multiprocessor, it is not globally coherent. That is, different L1 caches do not have to agree on what data currently exists at a given global memory address. For example, if a thread in block 0 writes to global memory, and a thread in block 1 then attempts to read from the same location *and* has an L1 *cache hit*, it will instead read from its own L1, which now contains stale data! This and similar global synchronization issues are relevant to the on-GPU tree construction algorithm (see Chapter 4, Section 5.2). L2 cache, on the other hand, is globally coherent (all multiprocessors access the same L2). It is possible for the programmer to enforce a read from global memory, bypassing the L1 cache, by qualifying pointers as `volatile`. This keyword is also needed to prevent the compiler from caching global and shared memory data in registers.

In addition to the above — which is less of a problem on newer hardware, where there effectively is no L1 cache — CUDA gives no guarantee about the *ordering* of memory accesses. Suppose a thread in block 0 writes to location i, and then to j; even if a thread in block 1 reads from j and receives the *new* data, there is no guarantee that a following read from i will also return new data! There exist memory fence functions specifically to overcome this situation: a thread waits at a memory fence to ensure that all writes *before* the fence are observed, by all other threads, as occurring before all writes *after* the fence. A performance penalty is typically incurred, since the fence forces threads to stall where they otherwise might not.

4.2 On-GPU Bounding Volume Hierarchy Construction

Bounding volume hierarchy (BVH) construction on GPUs has recently received much attention in the literature. The first such paper is that of Lauterbach et al. (2009),

and their so-called Linearized Bounding Volume Hierarchy (LBVH) forms the basis of the BVH construction method used in GRACE (see Chapter 4, Section 5). The construction performance was improved on by Pantaleoni and Luebke (2010), the hierarchical linearized bounding volume hierarchy (HLBVH) method, and a fast HLBVH implementation was presented by Garanzha et al. (2011). All of the above algorithms build the tree in a top-down manner; they begin at the *root node*, which contains all volume elements, and form successive, lower levels of *child nodes*. As a node's axis-aligned bounding box (AABB) is the union of its child nodes' AABBs, computing AABBs is an inherently bottom-up process. AABB computation must hence occur after hierarchy construction, and is typically done iteratively, level-by-level.

Karras (2012) presented an algorithm which fully parallelizes the hierarchy construction, with each node being processed independently. AABBs are once again computed in a bottom-up fashion, but here this is done with a single kernel launch, rather than level-by-level (see Chapter 4, Section 6). Karras and Aila (2013) then showed that such a structure can be rapidly post-processed, changing the topology of the tree and essentially bringing its ray tracing performance to parity with that of SAH-optimized BVHs built on the CPU.

Most recently, Apetrei (2014) presented what is likely the ultimate simplification of LBVH (see Chapter 4, Section 5, as this is the basis for the implementation in GRACE), and construction happens in a bottom-up fashion. They refer to this as Agglomerative LBVH (ALBVH). Bottom-up building has the advantage that AABBs and hierarchy construction can occur simultaneously, unlike for top-down or fully-parallel (H)LBVH implementations.

Interested readers are invited to follow the above references, but, as this work connects somewhat disparate fields, presented below the are the essentials of LBVH. Because it is relevant to GRACE, a description of the ALBVH algorithm is presented instead in Chapter 4, Section 5

LBVH

The fundamental idea behind LBVH is to order all input data along a one-dimensional, or space-filling, curve. Examples include the Morton curve and Hilbert curve, both shown for the two-dimensional case in Figure 3.5. The Hilbert curve in particular may be familiar as a means of domain decomposition, being used in, for example, the simulation

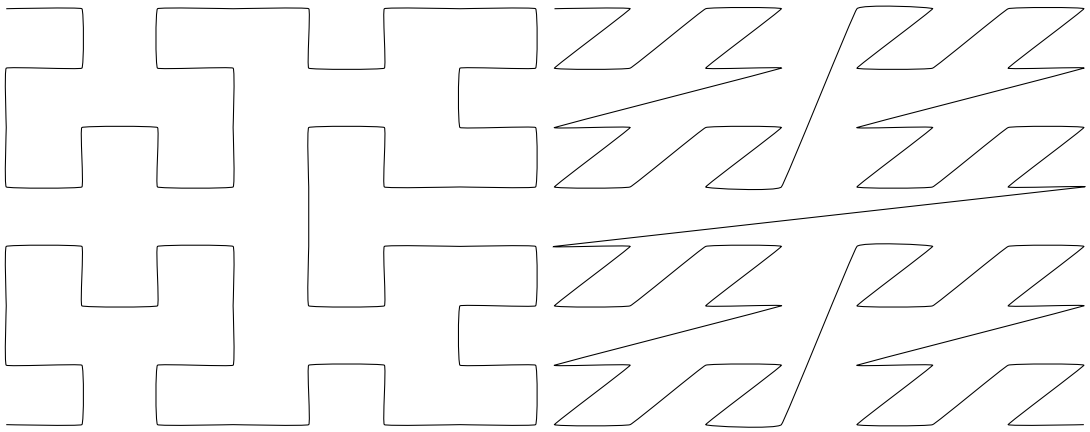


Figure 3.5. Left: Hilbert-key ordering of points on an 8×8 grid. **Right:** Morton-key ordering (z-order) of points on an 8×8 grid. In both cases, the curve begins at the point with the lowest-valued key, and proceeds in order of ascending key-value to the point with the highest-valued key.

codes GADGET-2 (Springel, 2005), GIZMO (Hopkins, 2015), and RAMSES (Teyssier, 2002). For both curves, each primitive has an associated integer key, derived from its centroid position, and primitives are sorted in ascending order of their key.

The root node of the BVH spans all keys, i.e. all primitives. To partition the root node, we look at the most-significant bit (MSB), say bit n, of the keys. All keys with a `0` nth bit are placed in the left child of the root node, and those with a `1` nth bit are placed in the right child. These two nodes are themselves split at the $(n-1)$th bit, and their children the $(n-2)$th bit, proceeding recursively until a node contains only one key, i.e. one primitive. (A node containing only one primitive is a *leaf node*, or simply a *leaf*. A node which instead contains other nodes is an *inner node*, often simply *node*.)

Nodes can thus be uniquely identified by their bit-prefix. For example, the left child of the root node has bit-prefix `0`, its left child has prefix `00`, and its right child has prefix `01`; the right child of the root node has bit-prefix `1`, its left child has prefix `10`, and its right child has prefix `11`. A given node contains *all* keys which share its bit-prefix.

In reality, it may be that the current MSB, say bit m, is identical for all keys in the node. In that case, the MSB becomes $m-1$, then $m-2$, $m-3\ldots$ until at least one key differs from the others at the most-significant bit. This ensures that every node has exactly two child nodes, but allows for the prefix-lengths of nodes at the same *level* (the same distance from the root node) to be unequal. It is also possible that one child may be a leaf node, and the other an inner node.

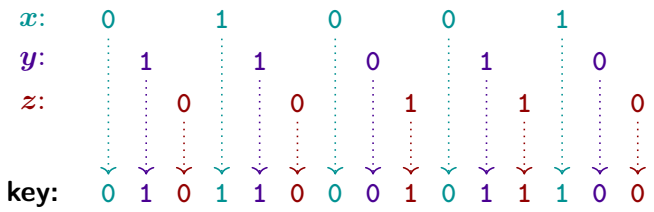


Figure 3.6. Computing a Morton key from three integer co-ordinates $(x, y, z) = (9, 26, 6)$. Their binary representations, (`01001`, `11010`, `00110`), are shown. This example has co-ordinates in the range $[0, 2^5)$, resulting in an effective grid resolution of $(2^5)^3$, or $32,768$ distinct points. Bits are interleaved to form the 15-bit key (3 co-ordinates with 5 bits per co-ordinate).

The split point in LBVH thus depends on the method used to generate the keys, but in general will likely be some approximation to the spatial median of the primitives in the node. Morton keys (see Section 4.2) are the most common choice because they can easily be computed in parallel. It is clear from Figure 3.5 that the Morton curve suffers from discontinuities, but as shown in Chapter 4, Section 7.3, use of the higher-quality Hilbert key offers no improvement in ray tracing performance. Further, the ALBVH algorithm implemented for GRACE provides a way in which we may partially mitigate such spatial discontinuities (see Chapter 4, Section 5).

Finally, as an interesting historical remark, note that tree construction using Morton keys is not itself a new idea. In astrophysics, for example, it was suggested by Barnes (1986) for octree construction, and was used by Warren and Salmon (1993) in the HOT N-body code. (To apply the LBVH method to octree construction, instead of partitioning a node based on 0 or 1 in a given bit, one instead considers three bits at a time, for which there are 2^3 possibilities, i.e. 8 partitions per node.)

The Morton Key

To compute a Morton key for a primitive, the primitive must have a single position associated with it. An obvious choice is the *centroid* (the geometric centre), which is trivial for cuboid-cells and spherical particles, and readily computed for other geometries, such as triangles and Voronoi cells.

Computing a key from a position (x, y, z) then proceeds as follows.

1. Compute, or define, the minimum (a_x, a_y, a_z) and maximum (b_x, b_y, b_z) values for all co-ordinates in the dataset.

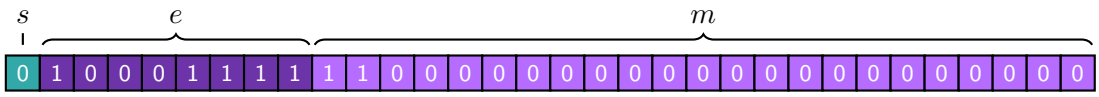


Figure 3.7. The bit layout of the number 114688 as a binary, single-precision floating point number in the IEEE 754 specification. s is the sign bit (where $s = 1$ identifies a negative number), e is the exponential component, and m is the mantissa, or fractional component.

2. Scale the co-ordinates to the range $[0, 1)$: for some x, we have $x' = (x - a_x)/(b_x - a_x + \epsilon)$, hence $x' \in [0, 1)$.
3. Map floating-point co-ordinates to integers in the range $[0, 2^m)$, where m is a tunable parameter: for some x', we have $\tilde{x} = 2^m \times x'$, hence $\tilde{x} \in \{0, 1, 2, \ldots, 2^m - 1\}$.
4. Interleave the bits of each integer co-ordinate to form the key. If the nth bit of co-ordinate $\tilde{x}$ is x_n, the key is $x_m y_m z_m x_{m-1} y_{m-1} z_{m-1} \ldots x_1 y_1 z_1$.

Steps 1 to 3 are equivalent to gridding all centroid positions, with the tunable parameter m defining the resolution of the grid: each axis on the grid spans $2^m - 1$ distinct points. Step 4 is illustrated in Figure 3.6 for $m = 5$.

In k dimensions, the key will contain km bits. Common choices in three dimensions are thus $m = 10$ and $m = 21$, producing 30- and 63-bit keys, and fitting into 32- and 64-bit integers, respectively. In GRACE, this choice is left to the user.

One potential optimization is to forego the mapping to an integer, item 3 in the above, and instead use the bits of the `float` representations of the input co-ordinates directly, discarding least-significant bits of the mantissa. Provided that all input co-ordinates are positive, Morton keys constructed this way will result in an identical order when sorted (excepting ambiguous cases where multiple keys have identical values). This is possible because of the IEEE 754 binary floating-point format, shown in Figure 3.7 for a single-precision value.

The value of a single-precision number is computed as

$$\text{value} = (-1)^s \times \left(1 + \sum_{i=1}^{23} m_{23-i}\, 2^{-i}\right) \times 2^{(e-127)}, \tag{3.15}$$

where s is the value of the sign bit; m_i is the value of the ith mantissa bit (with m_0, the right-most value in Figure 3.7, being least significant); and e is the value of the exponential, an unsigned integer.

First, note from Eq. (3.15) that for two positive values u and v, $e_u > e_v \implies u > v$

regardless of the values of m_u and m_v. Similarly, if $e_u = e_v$, then $m_u \geq m_v \implies u \geq v$ Thus, if $u \geq v$ and both are positive, the inequality relation is preserved in a lexicographic ordering of their bit representations.

5 Summary

Here, we have covered a significant quantity of background literature in order to help inform algorithm design, implementation and optimization in the forthcoming chapters. A comprehensive understanding of all covered material is not strictly necessary, but the critical points are summarized here.

First, Section 2 described the fundamentals of ray tracing, and the task of intersecting a line with both an axis-aligned bounding box (AABB) and a sphere. In Section 3.1 the concept of an acceleration structure was introduced, a data structure which vastly improves the efficiency of finding said intersections. Of these, the bounding volume hierarchy (BVH) should be understood going forward, with its object-partitioning, rather than spatial-partitioning (as in k-d trees) design. Graphics processing units (GPUs) were introduced in Section 4, and a basic understanding of the hardware and programming model is essential. The more technical points of that section will be revisited when discussing code optimization. Finally, and returning again to acceleration structures, Section 4.2 presented the core ideas underlying massively-parallel BVH construction. However, this topic is also given substantial attention in the next chapter.

4

GRACE: A GPU-accelerated ray tracing code

1 Introduction

GRACE[1] (loosely, GPU accelerated ray tracing) was initially developed as a ray tracing library for SPH datasets. It has been explicitly designed to run on NVIDIA CUDA GPUs. This chapter begins from this perspective, first covering the early design decisions influenced by the use of GPUs (Section 2), and by characteristics inherent to SPH simulations (Section 3). Section 4 introduces parallel primitive operations, and covers implementation choices. In Section 5, I describe the core algorithm used for construction of the acceleration structure, and its initial implementation. Some performance data is also included, which guides the optimization effort presented in Section 6. Section 7 first puts forward methods for generating ray distributions most useful in an astrophysical context; it then moves on to the traversal implementation, and its optimization. Verification of the correctness of the implementation is presented in Section 8 for SPH datasets. In Section 9, I discuss the performance of GRACE over various SPH datasets of differing particle counts and redshifts, with comparisons to alternative codes and libraries. Finally, in Section 10, I describe how I have generalized GRACE to allow for ray

[1] As presented here, available at `https://bitbucket.org/spthm/grace-devel`. It currently lacks documentation and a licence, but a release suitable for the astrophysics community is planned for the near future.

tracing arbitrary volume elements. This also contains further performance comparisons and optimization of the traversal implementation specific to non-SPH datasets.

Note that though much of this chapter assumes ray tracing of SPH datasets, 'particle', 'sphere', 'volume element', 'simulation element' and 'primitive' are used interchangeably. Only in Section 10 do the latter three explicitly refer to volume elements which are not, or may not necessarily be, SPH particles.

GRACE is a *template* library, and all code resides in header files. This requires use of C++, which at present is not particularly common within the astrophysics simulation community. For this reason, the API has been written with a focus on simplicity. Use of templates, and hence C++, is necessary to achieve the flexibility discussed in Section 10; in brief, functions to compute the AABB and ray intersections may be provided by the user. Thus, GRACE supports arbitrary simulation elements if a few, small functions are first provided to it.

Templates are also the technique by which both the standard template library (STL) and Thrust[2] allow a user to pass a custom function to one of their algorithms (e.g. a custom comparison operator for a sort of user-defined objects). Users who are familiar with templates and *functors* (function objects) should find GRACE straightforward to work with.

Finally, note that every stage of the ray tracing *pipeline* has either been written from scratch for the GPU, or uses an extant GPU library. This includes ray generation, BVH construction, BVH traversal, and post-processing (e.g. sorting of the output data and summing to obtain cumulative results along each ray).

2 Considerations for GPU Code

As discussed in Chapter 3, Section 4, graphics processing unit (GPU) hardware is significantly different from central processing unit (CPU) hardware. In developing GRACE — and GPU codes in general — it is therefore necessary to approach both low-level optimization and overall algorithm design somewhat differently. In this section, I discuss how the hardware has guided development and algorithm choices.

Ensuring that warp divergence is kept to a minimum requires a reduction in branching code. This is part of the motivation for choosing BVHs over k-d trees (see Chapter 3,

[2] A parallel algorithms library, `https://thrust.github.io/`. Also see Section 4.

Section 3.1 for a discussion of the differences). With a BVH, *traversal* (searching) is simple: we can simply run an exhaustive search over all nodes, which is accelerated due to the hierarchical structure of the BVH. Even in a naïve implementation, branching only occurs when one thread has to process a leaf, and another a node. The main disadvantage is that items will in general not be tested in-order along the ray; this point is further discussed at the end of Section 3.

Regular grids initially seem like a good candidate, their uniform layout and simple traversal properties being ideal for the GPU. However, aside from also exhibiting multiple-referencing, they do not adapt to the geometry of the dataset. Different grid cells may therefore have vastly different numbers of primitives within them, which is less than ideal for *load balancing* (in this context, ensuring that each warp in a block, and each block in a kernel, has an approximately equal work load). Note that scenes in computer graphics (CG) typically contain large, empty regions of the volume in which no primitives lie; this is an undesirable feature when using regular grids, but never occurs in astrophysical fields. Regular grids — or slightly more complex, multi-level variants — are therefore worthy of further research.

Memory allocation is also an issue on GPUs, in that we generally cannot allocate it within a kernel; instead, all memory must be allocated before launching a kernel.[3] Since BVHs have a predictable memory footprint (N leaves resulting in $N - 1$ nodes), memory allocation is relatively trivial. A final point in the BVH's favour (and possibly a result of the aforementioned points) is the existing literature: as was discussed in Chapter 3, Section 4.2, much of the work on GPU ray tracing in computer graphics has focussed on BVHs and, in particular, building acceleration structures on the GPU has focussed on BVHs.

The above comments, in conjunction with those in Section 3 — which apply only to SPH — have led to my choice of a BVH as the acceleration structure. Recent work by Vinkler et al. (2015) would appear to provide some empirical justification for this decision, though they do not find BVHs to be unconditionally more efficient than k-d trees for ray tracing, nor are their datasets and closest-intersection tracing algorithms necessarily reflective of the use-cases which primarily GRACE targets. Future work on this matter in the context of astrophysics would therefore be desirable.

[3] `malloc` and `new` can be called from within a kernel, but this entails managing per-thread pointers to memory, or communicating pointers between threads, neither of which are desirable when many threads are involved.

As for the ray tracing itself, the most natural solution is to map each thread to a ray. It is then important that, wherever possible, threads within the same warp request access to a contiguous region of memory; that is, their memory reads should be coalesced. In practice, this means that adjacent threads should follow the same, or very similar, paths through the BVH structure. This can be easily achieved using Morton keys (recall Chapter 3, Section 4.2).

Traditionally, data is passed to GPUs in a structure-of-arrays (SoA) format; n position vectors and radii would then be stored as in Listing 4.1.

Listing 4.1. Structure-of-arrays.

```
// SoA
struct particles {
  float x[n];
  float y[n];
  float z[n];
  float r[n];
};
```

However, CUDA allows for *vector loads* (and stores) of 8- and 16-byte elements in a single instruction. Vector loads increase the amount of memory requested by each thread per load instruction, and help utilize as much of the available memory bandwidth as possible. Vector loads and stores are guaranteed to be emitted by the compiler (excepting other optimizations it deems more appropriate) if using one of several built-in CUDA vector types, all of which are array-of-structures (AoS); an example is given in Listing 4.2. User-defined types may also be accessed in this manner, provided they have the correct alignment qualifiers. For a `float4`-like load or store operation, the type must be aligned to 16-bytes (or some larger power of two). This is *not* the case for a `struct` simply containing four `float` types, which would typically have the same 4-byte alignment as a `float`.

Listing 4.2. Array-of-structures.

```
// AoS. float4 is a built-in type, similar to below.
struct float4 {
  float x, y, z, r;
}
```

```
float4 particles[n];
```

Consider the case that two threads in a warp read from two consecutive particles in memory, and further that they require all four components of said particles for their computational task. For AoS, those threads request a single, contiguous block of 8×4 bytes, much better than the four separate requests to contiguous 2×4 byte blocks for SoA.

Finally, note that CPU-GPU data transfer times can be significant, relative to the execution time of GPU kernels. It is therefore important that as much computation occur on the GPU as possible. Even in cases where a task would be completed faster by the CPU, the ensuing data transfer(s) frequently result in a (possibly inefficient) GPU implementation of that task leading to faster overall execution.

3 Considerations for Ray Tracing SPH Datasets

While it will be demonstrated in Section 10 that GRACE is readily applicable to non-SPH datasets, the initial implementation specifically targeted SPH. This had an impact on my choices for algorithm design and lower-level details.

As far as ray tracing to find intersections is concerned, an SPH dataset is simply a set of overlapping spheres. To determine intersection, the latter method presented in Chapter 3, Section 2.2 was adopted (see Eq. (3.8) and Eqs (3.10)). An edge-case not covered previously is that of rays which begin or end within a sphere. A logical (though of course arbitrary) choice is described below.

First, consider the infinite line of which the ray, with its start and end points, is only a segment. Assume also that the ray begins or terminates within the sphere, but not both. The line must therefore both enter and exit the sphere. Finally, consider only the 2D slice of this sphere bounded by the great circle which passes through both the entry and exit points of the line. The line segment between the entry and exit points is then a chord of a circle, with length s. A ray which begins or terminates within a sphere is considered to intersect it only if the distance it covers within the sphere is $> s/2$. This is demonstrated in Figure 4.1.

For rays which begin and end within a sphere, the above works equally well provided $l > s/2$, where l is the length of the ray. When this is not the case, it may then be that very short rays are unable to intersect any particles. In GRACE— partly for simplicity

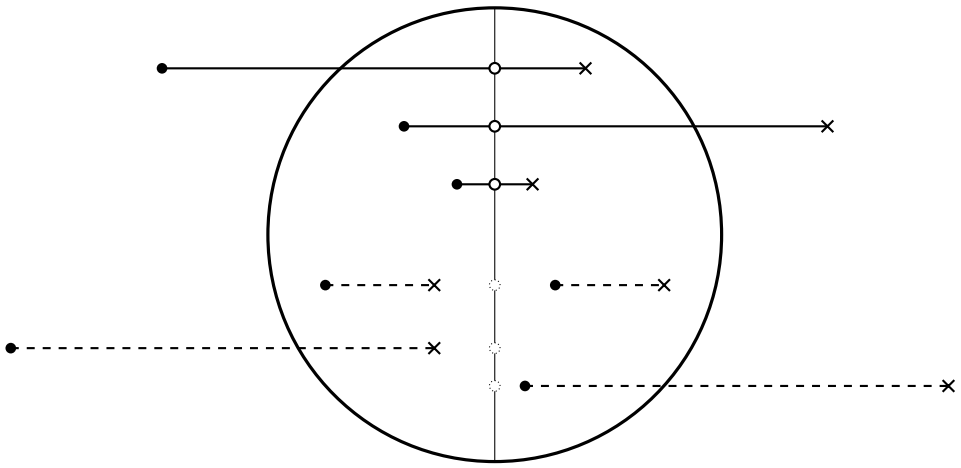


Figure 4.1. An example of several edge-cases in which rays either begin or terminate within a sphere, or both. All rays marked as solid lines are deemed to have intersected the sphere, whereas all rays marked as dashed lines are considered to have missed it. Ray origins are filled circles, and ray termini are crosses. Hollow circles denote the point of closest approach; dotted hollow circles indicate that a ray terminates before, or starts beyond, this point.

of implementation, and partly to avoid the above — all edge-cases are decided based on the distance to the point of closest approach, t in Eq. (3.8). In Figure 4.1, this is the distance from the ray origin to the point at which the line, of which the ray is a segment, intersects the marked diameter; the line-diameter intersection is marked with a hollow circle. A ray must have $0 \leq t \leq l$ for intersection, also shown in Figure 4.1; note that this is exactly Eqs (3.10).

A more interesting point is how to integrate a quantity along the ray, given that any point in the simulation volume is covered by multiple SPH particles. (As an aside, it is also interesting to note that many codes choose to project the SPH field onto a grid, or adaptive mesh, before performing radiative transfer (Oxley and Woolfson, 2003; D. Stamatellos and A. P. Whitworth, 2005; Alvarez et al., 2006; Razoumov and Sommer-Larsen, 2006; Finlator et al., 2009, e.g.). Only a few (Altay et al., 2008; Altay and Theuns, 2013; Forgan and Rice, 2010; Kessel-Deynet and Burkert, 2000; Pawlik and Schaye, 2008; Petkova and Springel, 2009; Susa, 2006) perform radiative transfer directly on the SPH field, fully preserving its resolution. For GRACE, I have adopted the approach of SPHRAY (Altay et al., 2008), as described in Chapter 2, Section 4.3. For one, this method preserves the native SPH resolution; for another, it readily maps to a generic ray tracing algorithm in which (all) ray-primitive intersection points are computed.

Finally, with regards specifically to SPH, it is not clear which acceleration structure is preferable (see Chapter 3, Section 3.1). Because multiple SPH particles will overlap a given point in space, a uniform grid or k-d tree will likely result in each cell or leaf referencing many spheres, resulting in extra work during traversal (increased computation time) and duplicates in the output list of intersected particles (increased memory usage). However, in their favour, it is relatively straightforward to traverse a k-d tree or a uniform grid in a closest-to-farthest manner. This is particularly useful when data at every ray-particle intersection is required, and when that data must be distance ordered.

BVHs do not suffer from the multiple-referencing problem, and trivially result in unique ray-particle intersections. However, the fact that SPH particles overlap necessarily requires that the AABBs in a BVH will overlap, which is not an efficient partitioning of the data. The greater the surface area of a node's AABB, the greater the probability of that node being intersected by a ray; hence, for a given set of data, one should attempt to minimize the surface areas of all AABBs (the principle behind the SAH, see Chapter 3, Section 3.2). Overlapping AABBs run contrary to this notion. That each node's contents may spatially overlap also prevents one from efficiently traversing a BVH structure in closest-to-farthest order; while the closest particle may be in node A, the next may be in node B, and the third again in node A, and so on. It is impossible to achieve such an order when considering each node individually, as is done in BVH traversal. If distance-ordered data is required, distance data must be saved for each intersection, and then all intersection data sorted by this value. Sorting n elements typically requires $O(n)$ temporary storage, resulting in memory usage equivalent to every ray-particle intersection being referenced exactly twice by a k-d tree or uniform grid traversal. Which method actually uses more memory in practice is thus unclear.

It is instructive to consider two common use-cases, one in which it is only necessary to *accumulate* a quantity along each ray, in any order (for example, total point-to-point column density), and one in which every ray-particle intersection must be saved, and in distance-order.

For the former, when using a k-d tree in a massively parallel environment like a GPU, it is not obvious how one would achieve this.[4] Pruning duplicates from an array

[4] Serial implementations, which trace one ray at a time, often store the identifier of the last ray to have intersected each particle. If a ray (re-)intersects a particle and finds its own index, it will skip that particle. Unfortunately, in an environment where multiple rays are traced simultaneously, this is not a solution.

of all ray-primitive intersections and then summing the quantities generated at each intersection would be a highly inefficient use of memory, and this increased memory usage would result in strongly *bandwidth* or latency limited kernel, reducing performance. The BVH is clearly preferable for this task, as data at each intersection may be accumulated as it is encountered, requiring only one output value per ray. This value may also be stored on-chip (in registers) for the duration of the traversal, and only written out, once, after the ray's traversal has ended.

For the latter, a k-d tree wastes only as much space as there are duplicates, and this might be partially or fully alleviated with intelligent bookkeeping during traversal. The increased memory usage of the BVH, necessary for the sort, is substantial, but it may also be trivially controlled: if each ray's intersection data is contiguous (see Section 4 and Figure 4.2, for example) then it is straightforward to only sort $m < N_{\text{rays}}$ at a time. This reduces the temporary storage required to $k\,O(m)$, where k is the mean number of intersections per ray, at the cost of a likely increase in execution time. Further, parallel sorts on the GPU are now very efficient (processing $> 10^9$ elements per second), with many implementations available, such as Thrust[5] and SGPU[6] (the latter is a derivative of MGPU[7] I have developed specifically for use in GRACE).

Parallel algorithms to remove duplicate entries also exist, and typically have much higher throughput than sorting, to the extent that the time spent post-processing the output of a k-d tree will likely be negligible.[8]

In summary, when all ray-particle intersection data is required, *and* when this data must be distance-ordered, k-d trees *may* have a slight advantage, but this is not certain, and in any case would require a non-trivial implementation. When only an accumulated per-ray value is required, which may be summed in any order, BVHs have the clear advantage. For these reasons, and those noted in Section 2, GRACE currently implements only a BVH acceleration structure.

[5] `https://thrust.github.io`

[6] `https://github.com/spthm/segmentedgpu`

[7] `http://nvlabs.github.io/moderngpu/`

[8] Compare for example CUB's `DeviceSelect` performance `https://nvlabs.github.io/cub/structcub_1_1_device_select.html` and its `DeviceRadixSort` performance `https://nvlabs.github.io/cub/structcub_1_1_device_radix_sort.html`.

4 Parallel Primitives

As noted in section Section 1, every stage of the ray tracing pipeline executes on the GPU. To support such an implementation, GRACE makes use of several parallel primitive operations. In this context, parallel primitives are conceptually simple operations, common to many different algorithms, which can be executed in a parallel manner. Examples include sorts, reductions, searches and prefix-scans. For readers not familiar with these tasks, they are briefly outlined in this section. I also discuss the chosen implementations.

A reduction over an array `A` with an operation $\circ$ and identity value `I` proceeds as described in Listing 4.3, and returns a single value.

Listing 4.3. A reduction.

```
reduction = I
for a in A:
  reduction = reduction ∘ a
return reduction
```

An inclusive prefix scan is similar, but returns an array `S` of equal size to `A`, as shown in Listing 4.4.

Listing 4.4. An inclusive scan.

```
reduction = I
for i in 0 ... len(A) - 1:
  reduction = reduction ∘ A[i]
  S[i] = reduction
return S
```

Listing 4.5. An exclusive scan..

```
reduction = I
for i in 0 ... len(A) - 1:
  S[i] = reduction
  reduction = reduction ∘ A[i]
return S
```

An exclusive scan is almost identical, but shifted-by-one, as shown in Listing 4.5. Note that `S[0]` is not well-defined for an exclusive scan, often being 0.

Efficient parallel sorts and scans are necessary for the ray tracing implementation in GRACE. However, often a single array containing logically separate data segments must be processed; for example, a single array which contains data for every intersection for every ray. Such a data layout is illustrated in Figure 4.2, where $\mathbf{d}_{ij}$ represents intersection j of ray i, and ray i has m_i intersections in total. In this case, each sub-array $\mathbf{d}_{i1} \ldots \mathbf{d}_{im_i}$ should be sorted, scanned or reduced as if it were the only input to the

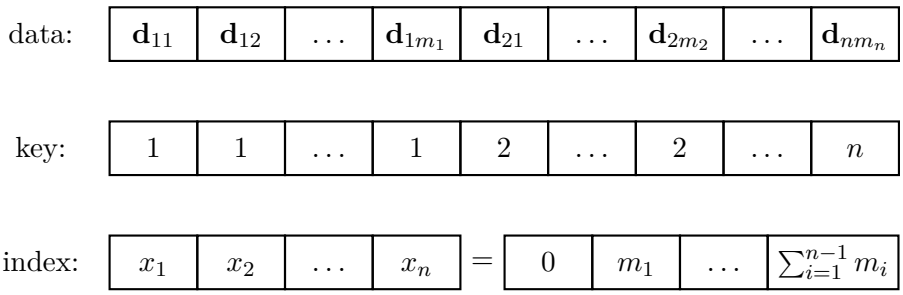


Figure 4.2. A typical layout of intersection data for n rays. $\mathbf{d}_{ij}$ represents data for the jth intersection of the ith ray, with ray i having m_i total intersections. The corresponding per-intersection keys required for a scan- or sort-by-key operation in Thrust are shown. The per-segment offsets required for a reduce-, scan- or sort-by-key operation in SGPU are also shown: x_i is the index in the data array at which $\mathbf{d}_{i1}$ is located.

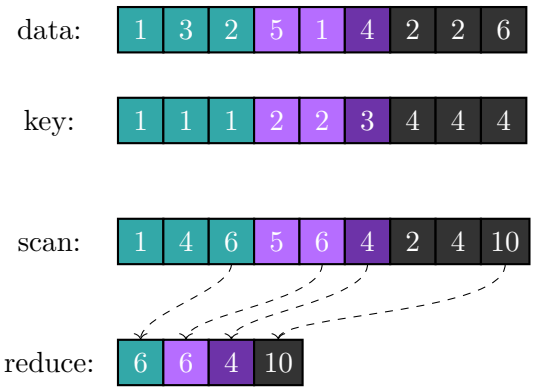


Figure 4.3. The input (data), segments (keys) and output (scan) of a segmented scan operation. A single input array contains logically-separate, but contiguous, sub-arrays. Segments have been colour-coded for clarity. Each segment is (inclusive) scan-summed.

sorting, scanning or reduction function. This is referred to as a *segmented* sort, scan or reduction.

Thrust offers scan- and sort-by-key algorithms, which accept two arrays of equal length, one of data and one of of keys. Each datum $\mathbf{d}_{ij}$ has a corresponding key value, also shown in Figure 4.2.

For scanning, data with consecutive-and-identical key values are scanned, resulting in independent scanning of each logical data segment, as required. This is exactly as illustrated in Figure 4.3, and is a parallel operation. Segmented reduction is achieved similarly in Thrust, though has reduced memory requirements as there is only one output value per segment. This is also illustrated.

Sorting is more involved, requiring two sort operations. First, all data and keys are sorted according only to the value of the *data*, e.g. distance along the ray. The

distance-ordered data and key values are then used as input to a *stable* sort-by-key. A stable sort guarantees that, when two keys in the sort compare equal, their relative order is preserved. Thus, while all data will be moved back into their original segments, the order within each segment will remain that produced by the first sort of the data, e.g. a sort by distance. This two-pass process is illustrated in Figure 4.4.

For a large number of intersections, storing a key value for each intersection has non-negligible memory requirements. Performance is quite poor relative to a non-segmented sort, due to the need for two sorts of both the data and the keys.

MGPU offers a more efficient approach, whereby only the index, x_i, into the data array at which the ith segment's data begins, is required; such indices are also shown in Figure 4.2. This offset can be computed as $x_i = \sum_{j=1}^{i} m_j$, where m_j is the length of the jth segment. One can therefore produce an array of all x_1, x_2, ... x_n values by performing an exclusive scan-sum over an array of segment-length counts, m_1, m_2, ... m_n. MGPU only provides functionality for sort- and reduce-by-key operations of this kind; I have implemented a scan-by-key based on the extant scan and reduce-by-key algorithms in the form of SGPU, an MGPU derivative.

In essence, the operation-by-key algorithms in MGPU and SGPU make use of the

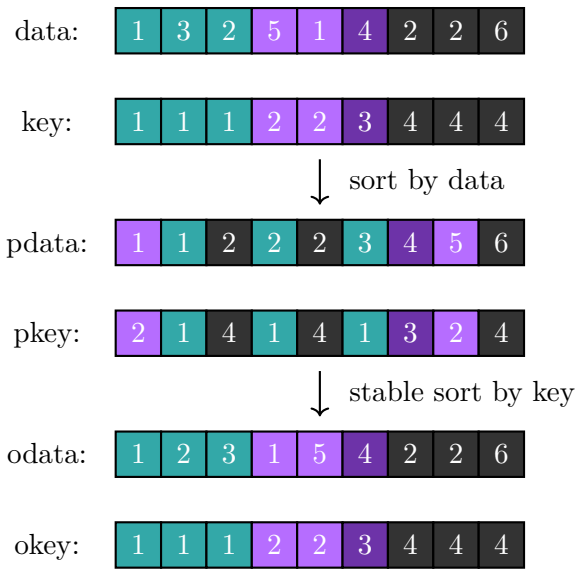


Figure 4.4. The input (data), segments (keys), partially-sorted data (pdata and pkeys) and output (odata and okeys) of a segmented sort operation. A single input array contains logically-separate, but contiguous, sub-arrays. Segments have been color-coded for clarity. In the first step, all data and keys are sorted only according to the input data. In the second step, all data and keys are stable-sorted according to the keys. This preserves the relative order from the first sort for elements with equal key values.

fact that, because all segments are contiguous, we need only know where they end and begin. The index of the segment in which any given value is contained is irrelevant. A key-like array of 0s and 1s, where 0 represents an even segment index and 1 an odd, is sufficient to efficiently identify segment boundaries. It also has the significant advantage that it may be compressed to a *bit array*, reducing storage requirements for the keys by a factor of 32, and similarly increasing effective bandwidth when reading keys (assuming an `int` type is 32-bits wide, as it is on CUDA GPUs). In practice, the key-like array is constructed in this compressed form directly from the segment indices x_i.

The performance and memory usage of these two approaches to processing segmented data are shown in Table 4.1. Accurately measuring the memory usage of Thrust's temporary memory allocations is non-trivial, so its stated memory usage includes only the input and output arrays. For SGPU, fewer segments will in fact reduce the memory usage, and more segments will increase it; in the extreme case that all segments have length one, input memory usage for Thrust and SGPU will be equal. Segment size also has an impact on performance, though the relative performance differences demonstrated in Table 4.1 remain. Finally, take note that, due to its requirement of one key per data item, all Thrust implementations must necessarily read (and write, for the sort) more data than SGPU implementations. This would lead to increased execution times for Thrust even if it was otherwise identical to SGPU. In point of fact, Thrust achieves higher performance than SGPU for non-segmented operations, and therefore appears to be significantly hampered by its one-key-per-datum design.

5 The Acceleration Structure: ALBVH

On-GPU BVH construction was a key design goal. Aside from reducing the CPU-GPU data transfer overhead, for some workloads a CPU-built acceleration structure is likely to dominate the execution time, making high-performance GPU ray tracing somewhat redundant. Of course, with spatially static data the acceleration structure may be built once and traced many times, amortizing the initial cost. However, for dynamic data this is not the case — for example, if radiative transfer and hydrodynamics are to be coupled, the BVH may need to be rebuilt (or at least modified) between every few calls to the ray-tracing kernel. It is also worth noting that while ray tracing typically scales only as $\log N$ or $N^{1/3} \log N$ in the number of particles, efficient BVH construction scales

Table 4.1. Comparison of the performance and memory usage of Thrust and SGPU for segmented reduction, scan and sort operations. In all cases, the input consists of 2×10^7 randomly-generated `float` values in randomly-generated segments of average size 500. Memory usage includes only input and output allocations for Thrust, and is a lower bound; for SGPU, all significant temporary allocations are also included. Mean of 10 iterations. All benchmarks were run on a Tesla M2090 GPU.

Segmented operation	Performance (M elements s^{-1})		Memory usage (MiB)	
	Thrust	SGPU[†]	Thrust	SGPU[†]
reduction	2500	12 000	153	84
scan[*]	1200	5400	229	153
sort	240	1100	153	153

[†]all SGPU code has been modified to run correctly on compute capability 2.x hardware for large input sizes, resulting in slightly reduced performance relative to the MGPU originals

[*]not present in MGPU

as N.

The choice of a BVH acceleration structure was motivated in Sections 2 and 3; the exact implementation was, however, not discussed. The intuition here is that, as particles in SPH datasets are relatively uniformly distributed (particularly when compared to CG scenes), the quality of the BVH is not of utmost importance. It is then permissible to forgo the complexities involved in constructing very high-quality BVHs on the GPU (see Karras and Aila (2013), for example), and I have opted for ALBVH, an LBVH (see Chapter 3, Section 4.2) variant. (The assumption that BVH quality is of lesser importance within the context of astrophysics is confirmed at the end of Section 7.3.)

Below, Section 5.1 describes my particular choice of ALBVH in an implementation-agnostic manner. Sections 5.2 and 5.3 present my initial implementation.

5.1 ALBVH Algorithm

ALBVH (Apetrei, 2014) construction begins with a list of n primitives and their associated Morton keys, sorted in ascending-key order, and proceeds in a bottom-up (leaves-to-root) manner. One primitive trivially corresponds to one leaf node, and in the below the terms are used interchangeably.

First, every leaf node computes its AABB and the node which is its parent. A parent node, knowing which leaves, and hence which primitives, it contains, can then

compute its own AABB and parent node. This proceeds recursively, until the root node is reached.

At the core of ALBVH lies a distance function,

$$\delta(i) \equiv \delta(i,\, i+1) \equiv \delta(i+1,\, i), \tag{4.1}$$

describing the distance between primitives i and $i+1$, where $i \in \{0,\, 1,\, \ldots,\, n-1\}$, i.e. zero-based indexing. For simplicity, δ here assumes $i+1$ exists, temporarily ignoring the special-case that $i = n-1$. The notation $\delta(i)$ is simply shorthand for the two-variable expressions, and consistent with Apetrei (2014), though henceforth the more explicit form is preferred.

The distance function determines how primitives should be grouped into nodes: primitive pairs with small distances are grouped together first (i.e. near the bottom of the tree), and pairs with large distances are grouped together later (i.e. near the top of the tree). This is illustrated in Figure 4.5, to which it will be useful to refer for the remainder of this section.

ALBVH defines the ith node to be the one which splits the ith and $(i+1)$th primitives. To clarify, denote the left and right children of some node i as L and R, respectively. Then the final primitive in node L is i, and the first primitive in node R is $i+1$.

The above rule identifies only the split-point of a node, while the tree also requires parent-child relationships. To illustrate, consider a primitive i. As described above, node i must split primitives i and $i+1$. Similarly, node $i-1$ must split primitives $i-1$ and i. Primitive i is therefore covered by both nodes $i-1$ and i; that is, both nodes are its *ancestors*. Only one of these ancestors may be the parent. The problem then is to determine, for a primitive i, which of these two possibilities is the parent.

In ALBVH , the solution is that a primitive always groups itself with the 'closest' neighbour, where closer is equivalent to a lower value for the distance function δ. For a primitive i, both $\delta(i,\, i-1)$ and $\delta(i,\, i+1)$ are computed. The smaller δ value identifies the split-point, and hence the parent node.

For example, suppose $\delta(i,\, i-1) < \delta(i,\, i+1)$; primitive i is closer to primitive $i-1$ than it is to $i+1$. Node $i-1$ is therefore the parent of primitive i, splitting primitives $i-1$ and i. Further up the tree hierarchy, node i splits the more-distant primitives i

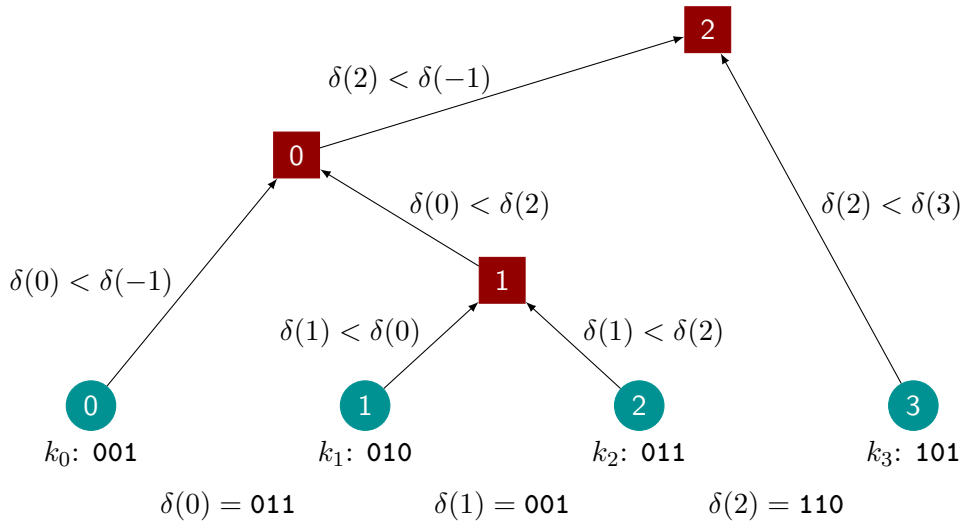


Figure 4.5. ALBVH hierarchy for four primitives with keys k_i. $\delta(i) \equiv k_i \oplus k_{i+1}$, where $\oplus$ is the bitwise XOR operator and $\delta(-1) = \delta(3) \equiv \infty$. Squares represent inner nodes; circles represent leaf nodes or primitives. The neighbour with the closest key, determined via δ, defines the parent index. For example, leaf 1 computes $\delta(0) = 001 \oplus 010 = 011$ and $\delta(1) = 010 \oplus 011 = 001$, finding $\delta(1) < \delta(0)$, hence node 1 is the parent.

and $i + 1$. This particular situation is demonstrated by leaf 2 in Figure 4.5.

Note that this relationship is not symmetric: primitive i may be closest to primitive $i+1$, and hence its parent is node i, but primitive $i+1$ may be closest to $(i+1)+1 = i+2$, and hence its parent is node $(i + 1)$. This situation is demonstrated by primitive 0 in Figure 4.5, and is not a problem. The result will be that the ith node's left child is leaf i, and its right child is a node which covers *at least* primitives $i + 1$ and $i + 2$.

The last piece of the algorithm dictates how to compute the parent of an inner node, rather than of a primitive or leaf. The construction is bottom-up, and so for any node being processed the range of primitives it spans is already known. Suppose, then, that a node covers the primitives $[i, j]$. If $\delta(i, i - 1) < \delta(j, j + 1)$, the node's parent is node $(i - 1)$, and the node is a right child. If $\delta(j, j + 1) \leq \delta(i, i - 1)$, the node's parent is node j, and the node is a left child. This is demonstrated for node 1 in Figure 4.5.

In practice, these comparisons take the form
`if` $\delta(i, i - 1) < \delta(j, j + 1)$ `{ ... } else { ... }`. Conveniently, this is robust in the case that multiple consecutive δ-values are equal, a property not shared by, for example, the fully-parallel LBVH algorithm of Karras (2012).

Finally, consider the distance functions $\delta(i, i - 1)$ for $i = 0$ and $\delta(i, i + 1)$ for $i = n - 1$, both of which must be well-defined to avoid special-casing these values of i.

Setting $\delta(0, -1) = \delta(n-1, n) \equiv \infty$ ensures that neither node -1 nor node $n-1$ will ever be chosen as a parent.

For pure LBVH, $\delta(i, i+1)$ is the bitwise exclusive-or (XOR) of the two primitives' keys,

$$\delta(i) \equiv \delta(i, i+1) \equiv k_i \oplus k_{i+1}, \tag{4.2}$$

where k_n is the key of the nth primitive, and $\oplus$ is the bitwise XOR operator. The resulting value will have its most-significant non-zero bit wherever the keys first (most significantly) differ.

However, a benefit of this distance-function approach is that it need not be based on the Morton key, and can be any commutative function. It could, for example, be based on the squared Euclidean distance between the primitives,

$$\delta(i) \equiv |\mathbf{r}_i - \mathbf{r}_{i+1}|^2, \tag{4.3}$$

where $\mathbf{r}_n$ refers to the position of the nth primitive; other choices include the total surface area or volume of the bounding box containing primitives i and $i+1$ (Apetrei, 2014). This is the mitigating factor alluded to at the end of Chapter 3, Section 4.2, diminishing the harmful effects of the discontinuities inherent in a Morton key ordering. In practice, a Euclidean distance metric ensures that two adjacent primitives a large distance apart will only be merged into the same node at higher levels in the tree. (This presupposes that the ordering of primitives expresses *some* spatial locality, so Morton- or Hilbert-sorting is still required.)

5.2 Initial ALBVH Implementation

Child node indices are required for BVH traversal, and are determined during construction, requiring non-temporary storage for at least two integral types. The parent index of a node is crucial during construction, but not needed during BVH traversal. Further — considering ALBVH specifically — indices of the left- and right-most primitives within a node are necessary for computation of the parent index, but the parent index itself need not be stored for reuse. Permanently storing these left- and right-most primitive indices within a node can be convenient, as they allow for inspection of the content of any node, as well as its size. While this may seem a relatively weak justification, these two values essentially come 'free' with the compact node layout presented in Section 7.3.

The node data layout is thus realized using a single `int4`-like `struct`, which is entirely compatible with the 16-byte vector load instructions discussed in Section 2.

For n primitives, δ values are pre-computed and stored in an array, `d`, such that `d[i] =` δ`(i, i+1)`. This array has size $n + 1$, and the edge-cases `d[-1]` and `d[n-1]`, discussed in Section 5.1, are present. For pure LBVH, computation of the distance function during construction would incur little overhead; for more complex metrics, such as Euclidean distance, pre-computing the values is an obvious optimization.

The implementation of Apetrei (2014) utilizes the bottom-up algorithm of Karras (2012), which climbs the tree from bottom-to-top in a single kernel launch; this has also been adopted for GRACE. To start, each leaf i (i.e. each primitive i) is assigned to a thread. That thread computes the AABB of the leaf, and computes the index of its parent, p, as described in Section 5.1. This index also identifies whether the current leaf is the left ($p = i$) or the right ($p = i - 1$) child of p. A left child writes its current index, i, as the left-child index of p; it also writes i as the left-most primitive index within p. A right child writes its current index as the right-child index of p; it also writes i as the right-most primitive index within p. This is illustrated in Figure 4.6.

Moving on to process the parent, a thread will first compute the AABB of p from the union of its two child AABBs, and then compute the parent of p, here denoted s, again as described in Section 5.1. If p is a left child, p is written as the left-child index of s, and the left-most primitive within p is written as the left-most primitive within s. If p is a right child, p is written as the right-child index of s, and the right-most primitive within p is written as the right-most primitive within s. This process continues, writing to the parent of s, and so on, and is demonstrated in Figure 4.6.

One may note that *exactly* two threads will write to each node. Of the two threads writing to a node, only the thread arriving *last* is permitted to continue the tree climb. This not an arbitrary choice: when the first thread arrives, only half of the parent node's data is present; when the second thread arrives, it is all present. The kernel returns after the root node is reached for the second time. In ALBVH , the root node may have any valid index, but can be easily identified because it is the only node to contain primitives [0, $n - 1$].

Detection of thread arrival-order is realized with an array of per-node counters, all initialized to zero. Every time a thread writes to a node, it increments that node's counter by one. To ensure correct behaviour in the case that two threads attempt to

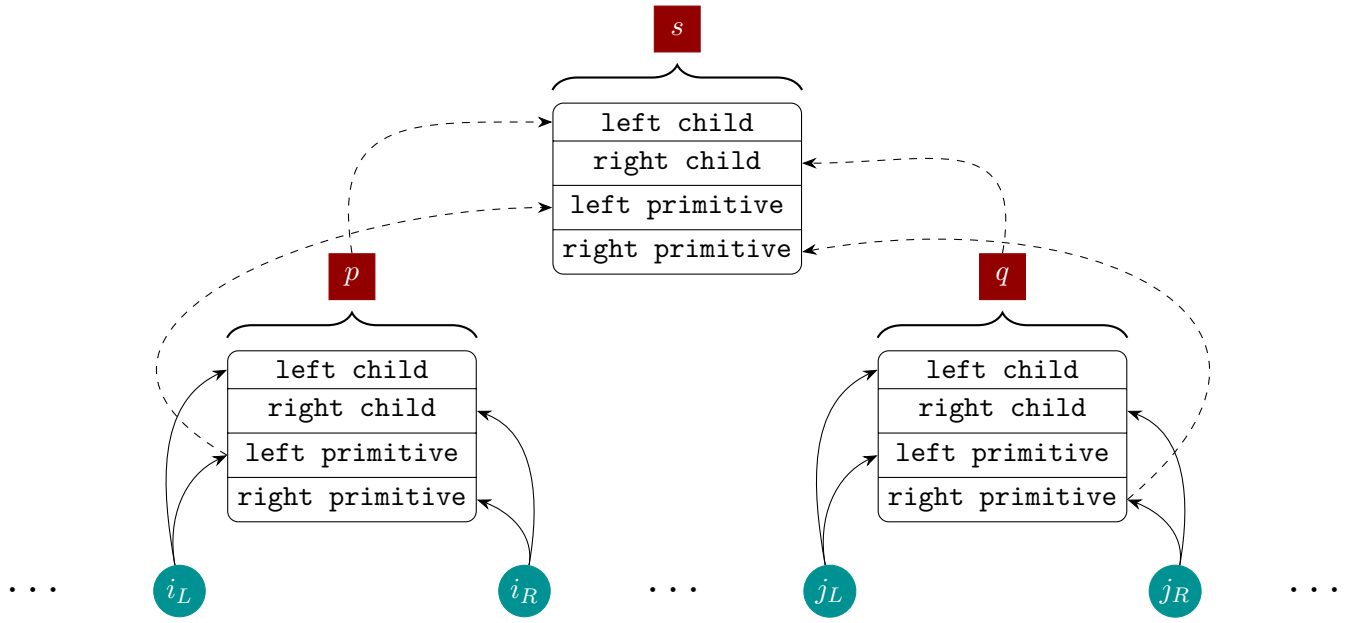


Figure 4.6. The propagation of node and primitive indices during the tree climb. Initially, four threads are processing leaves i_L, i_R, j_L and j_R. i_L and i_R are adjacent in memory, and j_L and j_R are adjacent in memory. Each thread computes its leaf's parent index, here either p or q, and writes to it. The *second* thread to write to p then computes the parent of p, here s, and writes to it. The *second* thread to write to q similarly finds the parent of q, here also s, and writes to it. The *second* of the threads to write to s will compute the parent of s, and so on. In all cases, the first thread to arrive at a parent idles for the remainder of the tree climb. Of the four threads active at the start, only one continues beyond s.

increment the value simultaneously, the increment is done via an `atomicAdd()` call. This is a CUDA builtin function which serializes all additions to a given location. It also returns the value which was stored before the increment. If this value is 0, the thread was the first to write to the node, and so returns. If this value is 1, the thread was the second to write to the node, and is allowed to continue the tree climb.

As an additional complication (and one which neither Karras (2012) nor Apetrei (2014) discuss[9]), CUDA has a weakly-ordered memory model. From the CUDA 7.5 programming guide,[10]

> *"The order in which a CUDA thread writes data to shared memory, global memory ... is not necessarily the order in which the data is observed being written by another CUDA or host thread"*

As a result, a thread might detect, via the counter, that a node has already been written to, but on loading that node it may in fact get stale, unwritten data.

Listing 4.6. C-like pseudo-code for the tree-climb. It is assumed that leaves are already processed, and that both `visits` and `i` are already defined from that procedure.

```
while (visits == 2) {
    Node node = nodes[idx];
    int pidx = node.parent_index();
    if (pidx == idx) {
        nodes[pidx].left = node.left;
        nodes[pidx].left_primitive = node.left_primitive;
    }
    else {
        nodes[pidx].right = node.right;
        nodes[pidx].right_primitive = node.right_primitive;
    }
    __threadfence();
    // atomicAdd() returns the value before the addition.
    visits = atomicAdd(&counters[pidx], 1) + 1;
    idx = pidx;
}
```

[9] Surprising, as use of `__threadfence()` is vital not simply for conformance to the CUDA memory model, but for correct behaviour in practice!

[10] For further information, see `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-fence-functions`

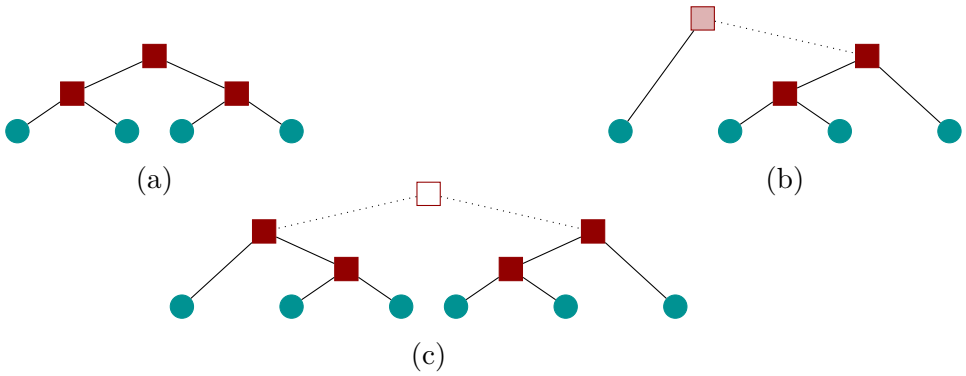


Figure 4.7. The three node conditions possible during a partial tree-climb, exiting when the number of primitives within a node is $> \phi_{\max} = 2$. Solid nodes have both child nodes written. Light-coloured nodes have only one child written. Hollow nodes have no children written, and contain no information. Dashed lines represent parent-child relationships not followed by any thread.

To avoid this, after writing to the parent, the builtin memory-fence function `__threadfence()` should be called, followed by the aforementioned `atomicAdd()`. Writes made before a call to `__threadfence()` are guaranteed to be visible to all other threads before any writes occurring after the call.

An outline of the tree-climb procedure for inner nodes is given in Listing 4.6. The similar leaf-to-parent step has been omitted, but prior to the loop of Listing 4.6, one should assume the following: the index `idx` has been set to the parent of a leaf, an inner node; that the leaf's index was written to said inner node; and that the value of `visits` has been incremented via an `atomicAdd()` call, prior to which `__threadfence()` was called. All threads reach the `while`-loop of Listing 4.6, but at most half will enter it, with at most half of those continuing for a second iteration, and so on.

5.3 Multiple Primitives per Leaf

The number of primitives present in a leaf can have a significant impact on the ray tracing performance. For this reason, the ALBVH implementation of Section 5.2 is modified to produce leaves containing (potentially) multiple primitives. A threshold, $\phi_{\max}$, is chosen such that the number of primitives spanned by a given leaf is permitted to be any integer $m \in [1, \phi_{\max}]$. These are henceforth referred to as *wide leaves*. The minimal data structure for a wide leaf requires two integers to specify the primitives it contains. This could be `int2`-like, compatible with the 8-byte vector load instructions discussed in Section 2.

A simple implementation might first construct a one-per-leaf tree and post-process it, marking nodes as invalid where appropriate, and converting nodes and leaves to wide leaves where appropriate. This would result in the set of valid nodes occupying a non-contiguous block of memory, wasting valuable global memory as well as being cache-unfriendly.

It would therefore seem preferable to first construct an array of contiguous wide leaves, and use them as a base for the inner nodes. We may achieve this via only partial construction of a one-per-leaf tree, in which threads climb only as far as necessary to ensure all wide leaves have been identified. These wide leaves are then written out to a contiguous block, which forms the lowest level of the final tree. Hence post-processing of the partial climb is still necessary, but the workload and memory requirements are nonetheless reduced.

The process begins with the tree-climb described in Section 5.2 and shown in Figures 4.5 and 4.6, but a thread will exit the climb after reaching a node containing $> \phi_{\max}$ primitives. At least one child of the first such node reached is guaranteed to be a wide leaf; this is demonstrated for $\phi_{\max} = 2$ in Figure 4.7. If only one child is a wide leaf, the other child must by definition contain $> \phi_{\max}$ primitives, and hence one of its descendants must be a wide leaf. For a pure tree-climb, only the primitives contained within each node are required; such a *pseudo node* is henceforth represented as a pair $[l, r]$, where l and r are the left- and right-most primitive indices, respectively. Once complete, all wide leaves may be detected as described by Algorithm 4.1.

For n primitives, tree construction proceeds as follows:

1. Allocate a temporary array, `P`, of $n - 1$ pseudo nodes, all initialized to an invalid state, [`null`, `null`].
2. Beginning with one thread per primitive, climb the tree as in Section 5.2, writing pseudo nodes to `P`. If the current node contains $> \phi_{\max}$ primitives, immediately exit the tree-climb.
3. Allocate an array, `L`, of n wide leaves, all initialized to an invalid state, [`null`, `null`].
4. Write all wide leaves to `L` as per Algorithm 4.1.
5. Remove invalid (unwritten) wide leaves from `L` using `thrust::remove_if()`.
6. The number of wide leaves, n', is the new size of `L`.
7. Allocate an array, `N`, of $n' - 1$ nodes.
8. Begin the tree construction in full, as in Section 5.2, using `L` as the leaves.

Algorithm 4.1. Writing all wide leaves to L, given a set of n pseudo nodes P. A leaf or psuedo-node is represented as its pair of left- and right-most primitives, $[l, r]$.

function WRITEWIDELEAVES(n, P, L)
  for $i = 0$ **to** $n - 2$ **do in parallel** ▷ m leaves $\implies$ $m - 1$ nodes
    $[l, r] \leftarrow P[i]$
    if $(l = \texttt{null}) \vee (r = \texttt{null})$ **then** ▷ Not fully written, Figure 4.7b, c
      $\texttt{size} \leftarrow \phi_{\max} + 1$ ▷ True size is *at least* $\phi_{\max} + 1$
    else
      $\texttt{size} \leftarrow r - l + 1$
    end if
    if $(l \neq \texttt{null}) \wedge (\texttt{size} > \phi_{\max})$ **then**
      $L[l] \leftarrow [l, i]$
    end if
    if $(r \neq \texttt{null}) \wedge (\texttt{size} > \phi_{\max})$ **then**
      $L[r] \leftarrow [i + 1, r]$
    end if
  end for
end function

In practice, `null` is implemented as -1;[11] 0 is not an appropriate choice because it is a valid value for l, but Algorithm 4.1 requires written and unwritten values be distinguishable. One may note that climbing only to nodes with sizes $\geq \phi_{\max}$ is sufficient; while potentially requiring one-fewer climb iterations, this method entails a slightly more complex version of Algorithm 4.1, as $\texttt{size} = \phi_{\max}$ is then an edge-case.

For efficiency, AABBs are not computed until the final step, where tree construction proper begins, henceforth referred to as *node building*. In the first stage of the node build, all the primitives covered by each leaf must be looped over to find the leaf's AABB; in all subsequent stages, the AABB of a node is the union of its child AABBs.

Prior to node building (item 8 in the above list) the array of δ values is compacted such that it can be indexed by the indices of wide leaves. That is, a new array, `cd`, of size $n' + 1$, is filled from the per-primitive array of δ values, `d`, as

```
cd[wi] = d[L[wi].right_primitive],
```
(4.4)

where `L` is again the array of wide leaves, `wi` is the index of a wide leaf, and `.right_primitive` refers to the index of the right-most primitive within that wide leaf.

[11] A negative value here precludes the use of unsigned integers, and thus limits us to maximum of $2^{31} - 1$ primitives. While forcing signed integers for only this purpose may seem inefficient, note that $2^{31} - 1$ `float4` values, e.g. for SPH particles, amount to 32 GiB, which exceeds the available memory of all current NVIDIA GPUs. The decision may, however, need revisiting in the future.

This satisfies `cd[wi] = δ(wi, wi+1)`. The edge-cases `cd[-1]` and `cd[n'-1]` are also preserved.

On the Fermi architecture (a Tesla M2090), the implementation described above performs relatively well. For a test SPH dataset of 128^3 particles at redshift $z \sim 11$ (see Section 9.2), with $\phi_{\max} = 32$, the leaf-building process completes in ~ 21 ms, and node building takes ~ 3 ms. (Sorting primitives by their Morton key, using the Thrust library, takes ~ 12 ms.) On Kepler hardware (a Tesla K20), performance is predictably improved; in particular, Kepler introduced faster atomic operations for global memory. Leaf building time decreases to ~ 7 ms, and node building to ~ 2 ms. (Sorting is reduced to ~ 8 ms.) In both cases the time taken to compute and compact the array of δ-values is $\lesssim 0.2$ ms.

The value of $\phi_{\max}$ has an impact on the time taken for the leaf- and node-building stages: predictably, lowering it reduces the former and increases the latter, while increasing it has the opposite effect. This is as expected, as it essentially shifts the relative workload between the two stages. However, the total time taken across both leaf- and node-building is constant to within $\sim 10\%$ for $\phi_{\max} \in [4, 64]$, with the total time increasing outwith the range. Further, in Section 9, $\phi_{\max} = 32$ is found to be approximately optimal for ray tracing performance, and so this value is assumed for the remainder of this section.

At this stage, it would be useful to make some comparisons between the implementation presented here and those of Apetrei (2014) and Karras (2012), henceforth also referred to as A14 and K12, respectively. Unfortunately, both authors use different hardware, and I do not have access to either. Each GPU used, as well as its theoretical peak throughput[12] and memory bandwidth is given in Table 4.2.

All node building performance data is then given in units of the corresponding GPU's peak throughput value, and in units of its total memory bandwidth. This only allows for a very crude comparison between implementations. Further, it does not take into account architecture-specific features; such details will result in differing performance for the same task, even when memory bandwidth and peak theoretical throughput are equal. Nonetheless, without access to the implementations themselves, or to identical hardware, this is all that can be done.

[12] Peak theoretical throughput is here, and typically, calculated assuming each CUDA core computes a single fused multiply-add (FMA) operation (which is *two* floating point operations) per clock cycle. The value is then $2 \times$ clock rate $\times$ core count.

Table 4.2. Comparison of the peak theoretical throughput of various NVIDIA GPUs.

Author	GPU	μarch	GFLOPS	Bandwidth (GB s^{-1})
Karras (2012)	GeForce GTX 480	Fermi	1340	177.4
Apetrei (2014)	GeForce GT 745M*	Kepler	642	32
This work	Tesla M2090	Fermi	1331	177.6

*Multiple versions of the GT 745M GPU are available, with the exact specifications depending on the original equipment manufacturer (OEM); the lower memory-bandwidth version is given here.

More reasonably, one may assume that, for a given architecture, the tree build time scales proportionally to the number of primitives. This is readily verified as holding true, to within a few per cent, for the results of A14, as well as the hierarchy construction and AABB computation times of K12. For comparison across scenes, performance measurements in Table 4.3 are therefore given in units of number of input primitives processed per unit time per unit throughput, and number per unit time per unit bandwidth. The pure-LBVH XOR distance metric is used for this work, and for A14, congruous with K12. Note however that for this work I take $\phi_{\max} = 32$; neither K12 nor A14 specify an equivalent factor, and most likely it is 1. For this work, memory operations which occur within Thrust's sort function are the only ones to have been included.

Depending on the metric, the implementation described in this section performs similarly to, or a factor of ~ 2 worse than, that of Apetrei (2014, Table 1), and both perform substantially worse than that of Karras (2012, Table 1). The latter's short Morton key evaluation times suggest an approach similar to that covered at the end of Chapter 3, Section 4.2. Their sort-by-key times are markedly lower than is possible with Thrust; comparison to the stated performance of CUB's radix sort[13] further suggests that the primitives themselves are not being sorted, a viable alternative being to sort an array of indices, which may then be used to index an unmodified list of primitives in Morton-key order. This would introduce a level of indirection and reduce coalescing when accessing primitives to build the hierarchy; however, Karras' hierarchy generation and AABB computation are also fast. Without access to the implementation, I can only assume Karras (2012) utilizes extremely well-optimized kernels.

In any case, it is clear that the primitive-sorting step — which is significantly slower,

[13] See `https://nvlabs.github.io/cub/structcub_1_1_device_radix_sort.html`, though in particular performance of the GTX 580 when sorting (`uint32`, `uint32`) key-value pairs.

Table 4.3. Comparison of the performance of LBVH implementations. The included authors have not clarified if any CPU-directed memory operations are included. For this work, memory transfers and allocations within Thrust's `sort_by_key` function are the only ones included, and $\phi_{\max} = 32$. Performance is measured both in number of primitives per ms per unit throughput (GFLOPS) and number of primitives per ms per unit bandwidth ($\mathrm{GB\,s^{-1}}$). K12 refers to Karras (2012) and A14 to Apetrei (2014).

Model	Primitives/10^3	# primitives $\mathrm{ms^{-1}}$ / GFLOPS			# primitives $\mathrm{ms^{-1}}$ / $\mathrm{GB\,s^{-1}}$		
		K12*	A14†	This‡	K12*	A14†	This‡
Stanford Bunny	69	—	5.28	—	—	106	—
Conference Room	283	170	—	—	1290	—	—
Armadillo	346	—	12.2	—	—	245	—
Stanford Dragon	871	178	9.35	13.3	1350	188	99.4
Happy Buddha	1088	—	9.49	—	—	191	—
Turbine Blade	1765	200	9.28	13.7	1520	186	103
$z \sim 11$, $N = 128^3$	2097	—	—	42.5	—	—	319
$z \sim 12$, $N = 256^3$	16 777	—	—	44.4	—	—	332

*Morton key evaluation, sort-by-key, hierarchy build time and AABB calculation

†Sort-by-key and tree construction time (including AABBs); LBVH distance metric used

‡Morton key evaluation, sort-by-key and tree construction time (including AABBs); LBVH distance metric used

Table 4.4. Comparison of the performance of LBVH implementations, but not including the time taken to sort primitives. Otherwise identical to Table 4.3.

Model	Primitives/10^3	# primitives ms^{-1} / GFLOPS			# primitives ms^{-1} / GB s^{-1}		
		K12	A14	This	K12	A14	This
Stanford Bunny	69	—	19.4	—	—	390	—
Conference Room	283	368	—	—	2790	—	—
Armadillo	346	—	44.7	—	—	899	—
Stanford Dragon	871	329	31.6	38.7	2490	635	290
Happy Buddha	1088	—	31.7	—	—	637	—
Turbine Blade	1765	343	32.0	41.3	2600	642	309
$z \sim 11$, $N = 128^3$	2097	—	—	62.6	—	—	469
$z \sim 12$, $N = 256^3$	16 777	—	—	65.8	—	—	493

per-primitive, for triangles than for SPH particles — has a significant impact on the metrics for this work.

In light of the above, comparing hierarchy construction times but explicitly excluding the sort time may be a fairer comparison. This is given in Table 4.4. When compared against peak throughput, the K12 performance data are still an order of magnitude greater than either A14 or the implementation presented here. However, looking at performance per unit bandwidth brings the results of A14 to within a factor of a few of K12, which is much more reasonable. This is consistent with the tree-climb method being memory bound, as claimed by K12.

At this stage, it is clear that the proposed ALBVH implementation has — possibly substantial — room for improvement. To ascertain where efforts might be better concentrated, fractional timing data for each step of several test datasets are given in Table 4.5. For the SPH datasets, both leaf-building and sort times dominate the build process; they are addressed in the following two sections.

While the ALBVH implementation will only be optimized for SPH datasets, some triangle-specific aspects are worth observing here. First, note from Table 4.4 that even when sorting is excluded, the performance metrics for the triangle-model datasets are consistently a factor of ~ 1.5 lower (worse) than for SPH. Secondly, note from Table 4.5 that while node-building is relatively insignificant for SPH datasets, leaf- and node-building consume a similar fraction of the total time for the triangle models. The

Table 4.5. Fractional time taken for each stage of the above ALBVH implementation for various input data. Morton key generation, sort-by-key, computation of δ-values, identifying wide leaves and writing them contiguously, compacting the array of δ-values, and generating the full node hierarchy are given.

Model	Fraction of total build time					
	Keys	Sort	δ-values	Leaves	Compact δs	Nodes
Stanford Dragon	0.038	0.68	0.0019	0.18	0.0013	0.12
Turbine Blade	0.036	0.67	0.0017	0.17	0.0010	0.12
$z \sim 11$, $N = 128^3$	0.061	0.32	0.0050	0.52	0.0031	0.084
$z \sim 12$, $N = 256^3$	0.058	0.33	0.0059	0.52	0.0029	0.086

raw timing data show that leaf-building performance, in terms of number of primitives per unit time, is consistent across all datasets; however, node-building performance is substantially worse for the triangle models relative to the SPH datasets. This is in fact the cause of the performance disparity between dataset formats evident in Table 4.4.

The only difference between the build procedures of triangle-model and SPH datasets which does *not* affect the leaf build is the loading of primitives; it is done only by the node-build kernel (to compute AABBs). While an SPH particle is represented by a single `float4`-like `struct`, triangles are represented by three `float3`-like `struct`s. This immediately entails a three-fold increase in memory traffic for primitive-loading. Additionally, triangles are stored in an array-of-structures (AoS) format, preventing full coalescing of said loads. Finally, use of `float3` data-types may be intrinsically detrimental, as their 12-byte size cannot be optimally accessed using the 8- or 16-byte vector-load instructions.

6 Optimizing ALBVH Construction

The performance data in Table 4.5 showed that, for SPH datasets, building the array of wide leaves contributes most to the total runtime. It is thus the first target for optimization, covered in Section 6.1. In Section 6.2, some of the techniques proposed for improving the leaf-build are applied to node building.

6.1 Optimized Wide-Leaf Implementation

Atomic operations are faster when acting on shared memory than on global memory. For this reason, Karras (2012) use shared memory per-node visit-counters when all threads in a block are processing the same set of nodes. As noted by Apetrei (2014), a thread cannot know *a priori* which nodes are being processed by other threads in its block. However, when computing wide leaves, a thread climbs the tree up to only a well-defined limit: no further than a node spanning $> \phi_{\max}$ primitives. This provides a deterministic bound on the range of primitives any given block will process, and this information can be used to locate all wide leaves without any inter-block communication. Shared memory may then used for the per-node counters, and `__threadfence()` may be replaced with `__threadfence_block()`.

To illustrate the current situation, consider that every block covers some number of leaves $n_L > \phi_{\max}$; block b then contains leaves $[bn_L, (b+1)n_L)$. To prevent any inter-block communication, all threads in block b are forbidden from writing to inner nodes with indices outside the range $[bn_L, (b+1)n_L)$. Now, note that in Algorithm 4.1, partially or fully unwritten nodes are assumed to have a size $> \phi_{\max}$; hence, the largest-sized nodes which *must* be fully processed cover $\phi_{\max}$ primitives. Also recall that, a node spanning primitives $[l, r]$ has an index i satisfying

$$l \leq i \leq r - 1. \tag{4.5}$$

It then follows that any wide leaf whose primitives exist entirely within a block will be reached by two threads in that block, and thus be fully processed. However, any wide leaves whose primitives span block boundaries cannot be identified as such without inter-block communication. This is illustrated in the top of Figure 4.8.

This problem may be solved by overlapping the range of primitives processed by adjacent blocks; that is, each block b processes some of the primitives which are also processed by block $b+1$, as illustrated in the bottom of Figure 4.8. The overlap size must be sufficient that wide leaves resulting from all conditions shown in Figure 4.7 can be correctly identified by Algorithm 4.1. These conditions in turn rely on item 2 from the tree construction steps presented in Section 5.3, which can be re-expressed as follows;

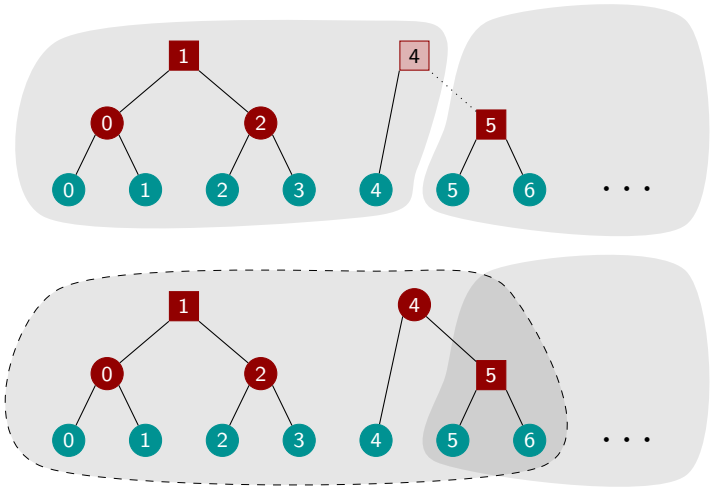


Figure 4.8. An example of wide-leaf building for $\phi_{\max} = 2$ and $n_L = 5$. Red circles are nodes which can be identified as wide leaves, and light-coloured nodes are those visited by only one thread. A dotted line represents a child-parent relationship which cannot be traversed. **Top:** each block processes only n_L primitives, resulting in an incomplete set of wide leaves. **Bottom:** each block b extends over the first $\phi_{\max}$ primitives within block $b+1$, resulting in a complete set of wide leaves, even in the worst-case illustrated here.

1. If a left child spans $\leq \phi_{\max}$ primitives, it must write its left-most primitive to its parent.
2. If a right child spans $\leq \phi_{\max}$ primitives, it must write its right-most primitive to its parent.

To determine the required overlap size, consider the general worst case: a wide leaf, index w, with size $\phi_{\max}$, the left-most primitive of which is the first primitive in block $b+1$, i.e. primitive $(b+1)n_L$, and which is the *right* child of its parent. The parent must therefore have index $(b+1)n_L - 1$, and so is not writeable from w's block, $b+1$. This is in fact the case illustrated in Figure 4.8.

To convince oneself that this is indeed the worst case, note that if the left-most primitive were farther leftward, w would not extent as far into $b+1$, reducing the overlap requirement on block b; if the left-most primitive in w were just one rightward, $(b+1)n_L + 1$, its parent would be $(b+1)n_L$, which again is writeable from block $b+1$; finally, if w were a *left* child, the parent would be $(b+1)n_L + \phi_{\max} - 1$, which is also writeable by block $b+1$ given our earlier stipulation $n_L > \phi_{\max}$.

Block b, which can write to w's parent at $(b+1)n_L - 1$, must then be granted sufficient overlap into the range of block $b+1$ that it can access all primitives within w. The left- and right-most primitives in node w are

$$\begin{aligned} l &= (b+1)n_L, \\ r &= (b+1)n_L + \phi_{\max} - 1, \end{aligned} \tag{4.6}$$

respectively. Comparing to block b's right-most primitive, $r_b = (b+1)n_L - 1$, we see that block b must extend to contain the first $r - r_b = \phi_{\max}$ primitives of block $b+1$. Similarly, block $b+1$ extends to contain the first $\phi_{\max}$ of block $b+2$, and so on.

To reiterate, in order for w to be identified as a wide leaf, it must be written to its parent, p. If w contains $\phi_{\max}$ primitives spanning $[l, r]$ ($r - l + 1 = \phi_{\max}$), then p is equal to either $l-1$ or r. For a block to have access to p, it must have access to all primitives within $[l-1, r]$.

Consider first the extreme case that l is the right-most primitive in block b, and that w is the left child of p,

$$\begin{aligned} l &= (b+1)n_L - 1, \\ p = r &= (b+1)n_L + \phi_{\max} - 2. \end{aligned} \tag{4.7}$$

This produces the right-most value that p may take whilst the range of primitives it spans is at least partially within block b. Block b clearly has access to l, but not r. The required overlap for block b is then $(b+1)n_L - 1 - r = \phi_{\max} - 1$.

Consider now the extreme case that p is the right-most node in block b, and that w is the right child of p,

$$\begin{aligned} p &= (b+1)n_L - 1, \\ l &= p + 1 = (b+1)n_L, \\ r &= (b+1)n_L + \phi_{\max} - 1. \end{aligned} \tag{4.8}$$

This produces the right-most value that w may take whilst the range of primitives spanned by p is at least partially within block b. Block b requires access to l and r, with r being larger. The required overlap for block b is then $(b+1)n_L - 1 - r = \phi_{\max}$.

Taking the larger of these two extremes, each block b must contain the first $\phi_{\max}$ primitives of block $b+1$. That is, block b must contain primitives $[bn_L, (b+1)n_L + \phi_{\max})$. From this overlap, we are guaranteed to reach every node which must be written in at least one block. No inter-block communication is necessary, and the atomically-incremented visit counters may therefore be placed in shared memory.

In the overlap region, threads in different blocks will write to the same nodes, but they will always write identical data. The only side effect is a small amount of redundancy. With the partial tree-climb complete, wide leaves are identified according to Algorithm 4.1, and tree-construction proceeds exactly as listed in Section 5.3.

It is in fact possible to remove the need for the memory fence operations altogether. Recall that, to climb the tree, only the left- and right-most primitives a node covers need be known. These can be communicated between threads using an `atomicExch()` to an array of parent-node left- and right-most indices. Provided said array is initialized with invalid data, if a valid end-index is obtained, we must be the second thread to arrive. This has the added benefit that the parent node is constructed from the result of the atomic operation — which is required in any case — and need not be loaded from memory. This is exemplified in Listing 4.7 (c.f. Listing 4.6 on page 83).

Listing 4.7. C-like pseudo-code for a tree-climb implementation which does not require memory fence operations. It is assumed that variables are correctly initialized before entering the loop.

```
while (other_end != INVALID_INDEX) {
    Node node(min(my_end, other_end),
              max(my_end, other_end));
    int pidx = parent_index(node);

    if (pidx == idx) {
        my_end = node.left;
    }
    else {
        my_end = node.right;
    }
    // atomicExch() returns the value before the exchange.
    other_end = atomicExch(&node_ends[pidx], my_end);
    idx = pidx;
}
```

Thus far, optimization has focused on reducing latency in the kernel, using faster memory where possible and reducing the use of blocking calls. Another potential route of optimization is to increase the ratio of active to inactive threads which are scheduled for execution. The tree climb naturally results in warps of only a few active threads, so

the goal is to continually funnel the block's (rapidly-reducing) workload toward as few warps as possible. This can be achieved by performing a block-wide scan-sum of the n_a active threads, thus enumerating them consecutively. The scan implementation I have used here is presented in Appendix C. Active threads then write their current state to shared memory, at the position of enumerated value. On the next iteration, the first n_a threads in the block — which necessarily implies the fewest possible warps — read the state from shared memory at their thread ID, climb the tree, and repeat the compaction procedure. Once a value $n_a \leq 32$ (the warp size) is reached, compaction ceases to be useful and is stopped. However, until that point, each warp that is scheduled for execution is wasting as few instructions as possible to idle threads. In principle this will reduce the number of times a warp is scheduled, on average, and hence the overall runtime.

It may also be interesting to question whether the leaves need be constructed directly from the tree structure at all. Certainly, a simple alternative would be to group consecutive runs of $\phi_{\max}$ primitives into leaves. Unfortunately, while Figure 4.10a shows the very short execution time expected, Figure 4.10b shows that it has a negative impact on ray tracing performance. In absolute terms, the increased traversal time exceeds the time saved during construction by a factor of ~ 2. This conclusion holds for all leaf sizes $\phi_{\max} \in [1, 128]$, with $\phi_{\max} = 32$, as was used in Figure 4.10b, providing approximately peak performance.

A compromise between a full tree-build and a simple consecutive-leaf grouping seems appropriate. To that end, first note that given a set of $\phi_{\max}$ consecutive primitives, there must be a wide leaf boundary either within the set, or between it and the set immediately following. A first step, then, is to process all $\lceil N/\phi_{\max} \rceil$ such groups, and identify the largest δ-value within each as wide-leaf boundary (where splitting immediately after a set's final primitive is also allowed). These are henceforth referred to as coarse splits. The maximum number of primitives which may exist between two consecutive coarse splits is $2\phi_{\max} - 1$, so clearly a further step is required. It is also worth noting that this is simply a segmented reduction, where the operator is the `argmax` function — that is, the `max` function which returns the index of the maximum value, rather than the value itself.

There would appear to be two solutions for ensuring all split-divided groups are sufficiently small. The splitting can be applied recursively: the largest δ-value between

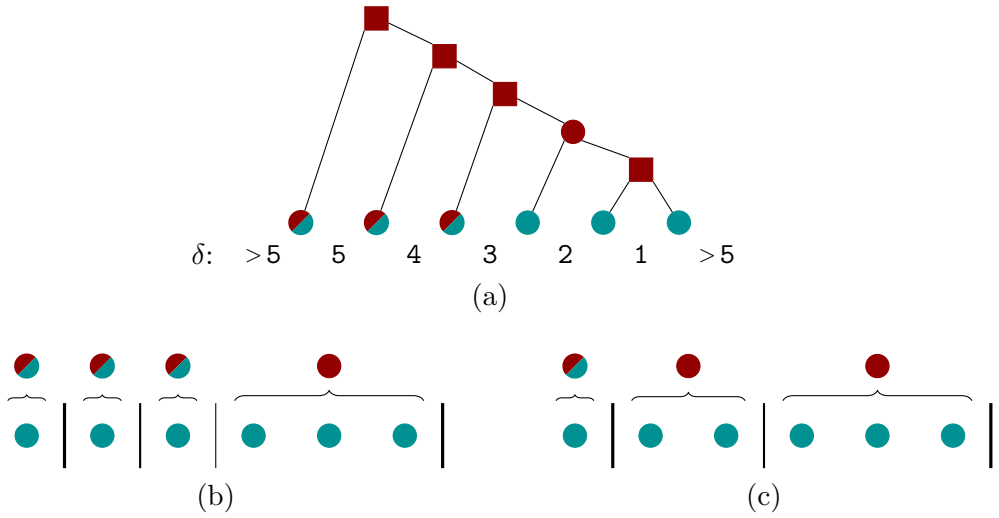
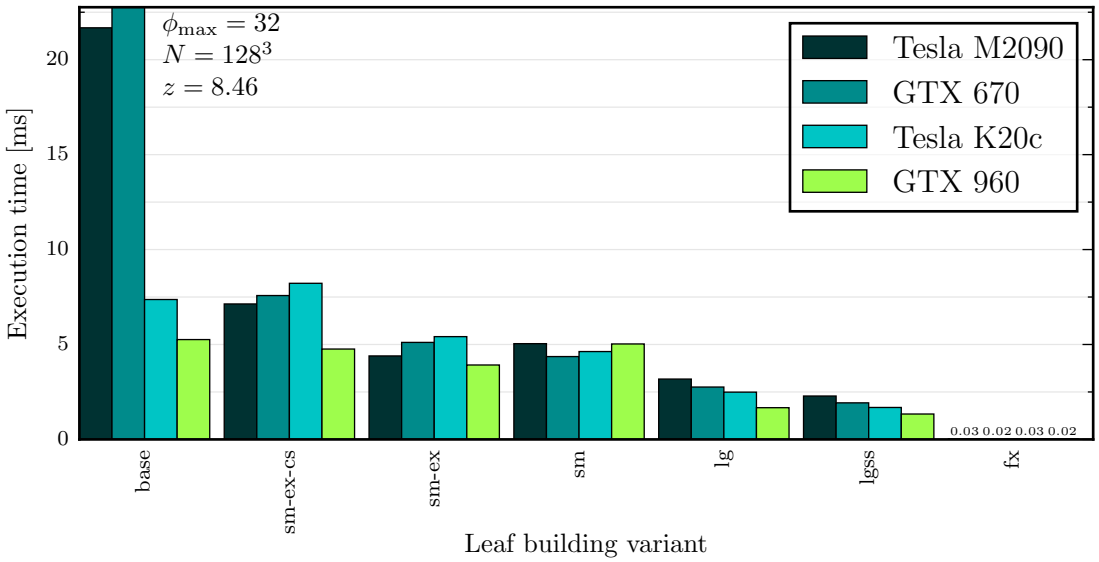


Figure 4.9. A comparison of wide-leaf build methods for $\phi_{\max} = 3$. Green circles are, equivalently, primitives or leaves, with ALBVH δ-values given between each pair. Red circles are nodes which will become wide leaves. Half-red half-green circles are leaves which will become size-one wide leaves. Vertical lines denote split positions, with thinner lines for later iterations. Shown is the wide-leaf structure obtained from a) a tree-based leaf build; b) a fully-recursive splitting, "lg" in Table 4.6; and c) a minimally-recursive splitting, "lgss" in Table 4.6.

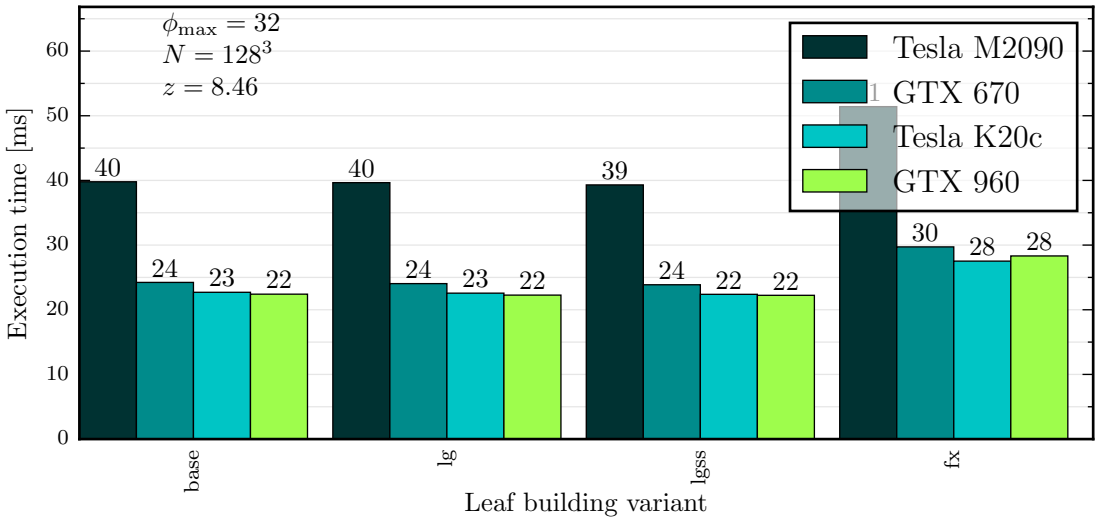
two coarse splits is chosen, producing two sub-groups, at least one of which must have a size $\leq \phi_{\max}$. The splitting procedure is again applied to the sub-group whose size exceeds $\phi_{\max}$, if it exists, and so on, until both subgroups have a size $\leq \phi_{\max}$. This is referred to as "lg" in Table 4.6 and Figure 4.10, and is demonstrated in Figure 4.9b.

Alternatively, note that in a run of $2\phi_{\max} - 1$ primitives, there exist exactly two split locations which produce two sub-groups whose sizes are both $\leq \phi_{\max}$. In a run of $2\phi_{\max} - 2$, there are three such locations, and so on. These potential locations may be searched for their largest δ-value, with it chosen as the only sub-split for that group. While this will likely select non-optimal splitting points, it is on average easier to compute. Further, it will avoid producing many smaller leaves followed by one large leaf in situations where both a tree-based leaf-builder and the above recursive-splitting method would do so. The latter point is visually demonstrated in Figure 4.9. This is referred to as "lgss" in Table 4.6 and Figure 4.10.

The leaf-build optimizations presented in this section are summarized in Table 4.6, along with a codified identifier. The performance of each leaf-build implementation is given in Figure 4.10a for a single GADGET-2 dataset. However, the results generally hold for datasets over the redshift range $3 \lesssim z \lesssim 21$, and for datasets containing $N = 256^3$ particles. An interesting observation is that times tend to increase, very



(a) Optimized leaf-build performance.



(b) Optimized leaf-build traversal performance.

Figure 4.10. Leaf-build time for the various implementations described in Table 4.6, and the resulting traversal times. Traversal here entails computing the number of particles intersected by each of 38 400 uniformly-distributed rays; all emanate from the centre of the input particle distribution and are of sufficient length to exit the simulation volume. Global memory operations are generally not included, unless they occur within a Thrust function, and with the exception of the wide-leaf compaction stage. The input file in both cases is identical, and a GADGET-2 snapshot.

Table 4.6. A summary of the leaf-build optimizations presented in this section. The codes given here are used in Figure 4.10.

Code	Feature	Description
base	—	Unoptimized, as described in Section 5.2.
sm	Shared memory	Shared memory is used for the target of the `atomicAdd()` or `atomicExch()` visit counters, which requires overlapping as described earlier in this section.
ex	`atomicExch()`	No memory-fence functions required, and parent node need not be read from global memory, as illustrated in Listing 4.7.
cs	Coalescing	After each child-to-parent climb, all threads in a block coordinate such that the smallest-possible number of warps are active for the next step (unless the number of active threads is ≤ 32).
lg	Leaf groups	Splits consecutive groups of primitives, recursively, until the largest sub-group contains $\leq \phi_{\max}$ primitives.
lgss	Leaf groups single split	Similar to above, but recurses at most once, while still guaranteeing that all sub-groups have size $\leq \phi_{\max}$.
fx	Fixed leaf groups	Consecutive runs of $\phi_{\max}$ primitives are assigned to consecutive leaves.

slightly ($\sim 5\%$ over the above range) as redshift decreases, i.e. as the input dataset becomes less uniform. Those implementations which produce differing wide leaves, and hence different acceleration structures, have their traversal performance given in Figure 4.10b. The time taken to count the number of intersections for each of 38 400 rays, all uniformly distributed, emanating from the centre of the simulation volume, and of sufficient length to exit the box, is shown. The traversal kernel itself is an optimized one, as will be discussed in Section 7.3.

On Fermi hardware, moving the targets of the atomic operations to shared memory, "sm", results in a factor of ~ 4 reduction in leaf-building time. On a K20, a Kepler-architecture device, a more modest reduction of ~ 1.5 is obtained. Kepler introduced faster atomic operations for global memory. And, by default, it does not cache accesses to global memory in the globally-incoherent L1 cache (see Chapter 3, Section 4.1), which means `__threadfence()` and `__threadfence_block()` are essentially identical. These two points likely account for the smaller gain. On the Maxwell-based GTX960, performance is essentially unaffected.

The GTX670 results are vastly improved; however, performance of the un-

optimized implementation appears to be spuriously poor. In particular, use of `__threadfence()` on this device results in significantly increased run-times; a change to `__threadfence_block()` — as is done for all other implementation present which require a memory fence call — rectifies this. The result is particularly surprising as, much like the K20, the GTX670 does not cache writes to global memory in its L1 cache, and one would therefore expect both operations to perform more-or-less identically. Some investigation suggests the observed behaviour is not expected; additionally, it seems that this device's use as the display card in its system has, for whatever reason, resulted in the extended runtime. Unfortunately, it was not possible to change this.

The overhead of enforcing a minimum number of active warps in each block, "cs", outweighs any efficiency gain across all hardware and input datasets.

Removing the need for the memory-fence operation is favourable on the oldest (M2090) and newest (GTX960) devices, but, unexpectedly, has a negative impact on both the GTX670 and the K20. Other than the — seemingly unlikely — possibility that an `atomicExch()` operation is substantially more expensive than an `atomicAdd()` on these devices, I am unable to offer an explanation for this result. Both kernels are, quite deliberately, near-identical; the "sm-ex" variant has the `atomicExch()`, no `__threadfence_block()` and no read of the parent node from global memory, but is otherwise identical to the "sm" kernel.

Performance of the non-hierarchical methods is more predictable. As already noted, using fixed-sized leaves is extremely fast, but traversal performance is disproportionately reduced. The single-split implementation, "lgss", builds faster than the fully-splitting variant, "lg", but also results in marginally faster traversal. It would appear that the more consistently-sized leaves, while perhaps not optimally grouped, are useful in practice. Most obviously, having fewer small leaves better amortizes the cost of looping over all primitives within an intersected leaf, making improved use of cached access to the primitives, since they are stored contiguously.

The "lgss" implementation, then, gives the net-best performance across construction and traversal for SPH datasets for all tested hardware. This is due to its simplicity of implementation and consistency of resulting leaf sizes.

6.2 Optimized Node Building

The optimized, tree-based wide leaf method of Section 6.1 can be adapted for construction of the nodes. This may be useful for datasets in which a larger proportion of the computational time is expended during the node build; for example, if the optimal value of $\phi_{\max}$ is much smaller than 32, or if the dataset covers a very large number of primitives. Also recall from Table 4.5 that the triangle-mesh datasets result in substantially longer node-construction times compared to similarly-sized SPH datasets, even at an identical $\phi_{\max}$.

In Section 6.1 it was shown that the ALBVH structure can be built up to a well-defined point without any inter-block communication. A logical extension is then to apply the procedure iteratively, with each iteration adding a new layer of nodes, until the BVH is complete.

The first iteration begins at the wide leaves. The nodes comprising the starting point of an iteration are henceforth referred to as *base nodes*. The tree-climb proceeds similarly to that described in Sections 5.2 and 6.1. Each block b covers all base nodes in the range $[bn_b, (b+1)n_b + \phi_{\max}^{\text{node}} - 1]$, for some $n_b > \phi_{\max}^{\text{node}}$, and hence overlaps the first $\phi_{\max}^{\text{node}}$ base nodes of block $b+1$. A thread stops climbing if it is the first to reach a parent node, if the parent has an index outwith $[bn_b, (b+1)n_b + \phi_{\max}^{\text{node}} - 1]$, or if the parent spans $> \phi_{\max}^{\text{node}}$ base nodes. The per-node visit flags are initialized to invalid values, and updated via an `atomicExch()`; similarly to Listing 4.7, information is communicated here — specifically, the left- or right-most base-node index of the current node.

After the iteration is complete, a work queue is populated with the indices of the nodes at which the next iteration should begin. The process outlined above is then repeated, beginning from these new base nodes.

Before considering how the work queue is populated, it should be emphasized that, in the context of the iterative implementation described here, a node's size refers to the number of *base nodes* it spans in a given iteration. This (i) is different from the node's size in terms of the wide leaves it spans (excepting the first iteration); (ii) will change between iterations; and (iii) is meaningless once a node forms a part of the completed hierarchy — that is, once it becomes a descendant of a base node. Similarly, the index of a node in a given iteration is expressed in terms of the base nodes: parent node p splits the hierarchy between base nodes p and $p+1$. The $[bn_b, (b+1)n_b + \phi_{\max}^{\text{node}} - 1]$ range given above refers to these base-node indices, rather than the actual node index.

After an iteration, every node which is an ancestor of the base nodes is in one of four states, analogous to Algorithm 4.1 for detection of wide leaves:

1. Both the left- and right-child data were written to the node, and it covers $> \phi_{\max}^{\text{node}}$ base nodes.
2. Both the left- and right-child data were written to the node, and it covers $\leq \phi_{\max}^{\text{node}}$ base nodes.
3. Only one side of the node was written.
4. The node was not written.

In the first case, the tree climb would have ended at the current node. The index of this node is therefore written to the work queue.

In the second case, the tree climb will have continued beyond this node, and so no action is taken; it is now a part of the completed hierarchy.

In the third case, the node is incomplete, so we should attempt to complete it in the next iteration. Exactly one of its children must be fully-written, with a size $\leq \phi_{\max}^{\text{node}}$; this child is added to the queue. The other child was either not fully-written, or it had a size $> \phi_{\max}^{\text{node}}$

In the final case, no action is taken.

The iterations end once the work queue contains only one node, which is guaranteed to be the root node, and the hierarchy is therefore complete. A two-iteration node build is illustrated in Figure 4.11.

Implementing the above presents several related problems. First, base node indices must be propagated during the tree climb in order for the climb-exit conditions to be tested. However, the tree structure itself must obviously still contain actual node indices. The solution here is to propagate the base-node indices, $[l_b, r_b]$, in shared memory. These two values represent the left- and right-most base-nodes covered by a given node, respectively.

Unfortunately, this presents a new problem: information about the size of a node, in terms of the base-nodes it spans, is lost after each iteration; the rules listed above for populating the work queue then cannot be applied. This is solved with a second tree-climb, identifying nodes to be added to the queue.

Before the first iteration, all nodes are assigned invalid, `null` (in practice, values of -1 are used) data. The queue-filling climb begins at the base nodes of the just-completed iteration. All threads then immediately climb to the parent of their base node. The

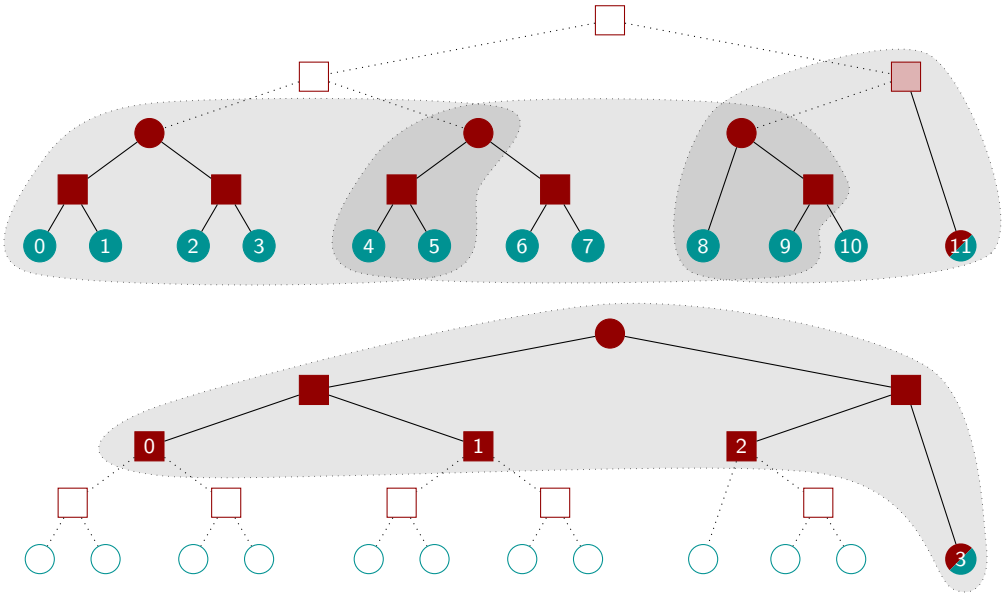


Figure 4.11. Two iterations of node building, with $\phi_{\max}^{\text{node}} = 2$ and $n_b = 4$. Shaded areas represent the nodes covered by a block, hollow nodes are those not processed in the given iteration, and light-coloured nodes have only one set of child data written. Base nodes have their base-node indices shown, i.e. their position in the work queue. **Top:** the first iteration. We begin with the wide leaves as the base nodes. Nodes which become base nodes for the second iteration are marked as circles, including wide leaf 11. **Bottom:** the second iteration. Base nodes are again marked with their consecutive base-node indices. The second iteration identifies only one base node, meaning the node building is complete.

parent has its state examined, which can be either fully written (solid squares and red circles in Figure 4.11), or partially written (light-coloured squares in Figure 4.11). In the second case, the thread writes the child it came from to the work queue, and then exits the climb. In the first case, one of the two threads currently examining the parent should continue the climb; this is chosen simply as the thread which came from the parent's left child. The procedure is repeated at the next parent, and so on. Note, however, that on continuing the climb a thread may encounter a third node state: a node which is fully unwritten (hollow squares in Figure 4.11). In this case it proceeds identically to a partially-written node, writing the child it came from to the work queue and exiting. The above rules for processing a parent can be expressed as in Algorithm 4.2.

Algorithm 4.2. Parent-processing algorithm for filling the work queue. Adds a node to the work queue if necessary, and returns true if the thread should exit the tree climb, else false.

```
function ProcessParent(child, parent, queue)
    N_child ← (parent.left ≠ null) + (parent.right ≠ null)
    if N_child < 2 then
        queue[thread_id] ← child
        return true
    else if child ≠ parent.left then
        return true
    else
        return false
    end if
end function
```

To ensure that all writes to the work queue target a unique location, a thread always writes to a location offset by its (unique) thread ID. In general, this will result in a non-contiguous queue. The queue is therefore compacted, removing all unwritten elements. This is easily achieved by resetting the queue to contain only invalid (e.g. -1) indices before it is filled. After populating the queue, a call to `thrust::remove()` removes the invalid indices. Consecutive items in the queue are then easily assigned to consecutive threads, and all threads are active at the beginning of each iteration's tree-climb.

The node-building performance of the above-described implementation is shown in Figure 4.12. For the smaller, $N = 128^3$ dataset, this iterative scheme results in worse performance relative to the implementation of Section 5.2 ("base"), with the exception of the GTX670, which again gave spuriously poor results for all implementations requiring

a `__threadfence()`. However, results are improved when moving to the larger $N = 256^3$ dataset, which has many more nodes. Only the M2090 and K20 devices have sufficient memory to build the BVH for this larger dataset. Increasing the number of nodes required for the $N = 128^3$ dataset by setting $\phi_{\max} = 1$, i.e. size-one wide leaves, does not change the conclusions drawn for that dataset.

In light of these mixed results, a second implementation is devised, referred to henceforth as *join queues*, and "jq" in Figure 4.12. The goal is to forego the separate queue-filling step necessary for the iterative method, while maintaining a high ratio of active-to-inactive threads. All threads begin at a base node, and propagate indices and AABBs as necessary to the parent. Note that the per-node visit flags must be in global memory here. The second thread to write to a parent adds it to the join queue — that is, a queue of nodes which have just been 'joined', or fully written. Within each block, all threads wishing to write to the join queue are enumerated via a block-wide scan sum, as in Appendix C. For n such joining threads, the first thread in the block atomically increments a global counter as `r = atomicAdd(counter, n)`. The value returned by the atomic operation is propagated to all threads in the block via shared memory. Each joining thread then writes to the join queue at the location `r + thread_offset`, where the offset is the thread's scan-sum value. Thus the join queue is filled contiguously without need for a separate kernel launch, and without a call to `thrust::remove()` to remove empty elements. Do note, however, that in general the join queue is filled with non-consecutive node indices.

As implemented here, a separate kernel performs the first wide leaf-to-parent climb, computing AABBs from the primitives within each leaf. All subsequent kernels can then read AABBs from their starting node, and never have to process any primitives. This differs from the implementation in Section 5.2, where a single kernel is used, which checks whether a node is an inner node or a wide leaf and computes the AABB of said node appropriately.

Performance for the join queue method is typically worse than for the iterative scheme, with the single exception of the K20 device and $N = 256^3$ particle dataset.

An obvious extension to this implementation is to allow threads to climb farther than a single parent before writing to the join queue. This is also implemented, displayed as "jqc" in Figure 4.12. A threshold of 10 child-parent climbs was chosen as the limit here. Varying the threshold results in changes of $\lesssim 5\%$ to the results. However, increasing the

threshold generally increases performance. For this reason, a full climb implementation was also tested, "fc" in Figure 4.12. It is functionally equivalent to a climbing join-queues implementation in which the climb limit is greater than the maximum depth of the tree. In reality, the (small) overhead of checking and enforcing the climb limit is removed. Additionally, the per-node visit flag is again used to propagate the left- or right-most wide leaf in the current node. Other than this, the full climb is very similar to the implementation of Section 5.2, "base", except — just as for the join queues — it launches one kernel to compute all wide leaf AABBs and write them to their parent nodes, then a second to build the remainder of the hierarchy. The first kernel is identical to the one used in the join queues method. The second kernel therefore also differs from that of "base" in that the nodes input to it are, in general, not consecutive.

As one may have already predicted, the full climb method outperforms both the join queues and the climbing join queues implementations. The only exception is the GTX960, for which climbing join queues actually achieves highest performance. This apparent outlier was not further investigated, though it may be indicative of performance characteristics of newer architectures.

Finally, all schemes suggested so far may be updated to make heavier use of shared memory and to minimize the number of active warps in each block, as was done in Section 6.1. Recall that for the iterative implementation, no inter-block communication is required. It is therefore straightforward to cache left- or right-most base node indices and AABBs in shared memory. In addition to writing this information to the parent node, in global memory, a thread also writes it to shared memory at a location related to its own thread ID. The per-node visit flags, which are also located in shared memory, are initialized with invalid values. Each thread updates said flag via an `atomicExch()`, writing its own thread ID. On having valid data returned from such an operation, a thread knows that it is the second to write to the parent node, and should continue climbing. It also obtains the thread ID of the other thread to write to this parent. It can therefore read from the location in shared memory at which the other thread cached its data, eliminating the need for a read of the parent node from global memory.

The other methods follow a very similar procedure. However, their visit flags are necessarily stored in global memory. There is also no guarantee that the two threads writing to a given parent node will be in the same block. On obtaining a valid value from the `atomicExch()`, a thread first determines whether the returned thread ID

corresponds to a thread in its own block. If it does, the shared memory cache may be read. If it does not, a read from global memory is still necessary.

For the climbing join queues implementation, the situation is further complicated. Not only must both threads writing to a parent be within the same block, they must also have written to the parent during the same kernel call. Here, both the thread ID and the current iteration — which is passed to the kernel and tracked on the CPU-side — are packed into a 64-bit `long long int`, and this is written to the visit flags via an `atomicExch()`. On obtaining valid data, a thread must confirm both that the thread ID corresponds to its block, and that the iteration matches the current iteration, before accessing shared memory. The use of this wider atomic operation is likely the reason that the shared memory optimization is generally not beneficial for the join queues implementation, as evidenced in Figure 4.12, variant "jqc-sm".

Minimizing the number of active warps is achieved identically as described in Section 6.1.

The performance of all above implementations is presented in Figure 4.12, with the codified identifiers for each implementation listed in Table 4.7. No implementation performs best across both datasets and all hardware.

For the M2090, the iterative method achieves the highest performance for $N = 128^3$ and $N = 256^3$ at $\phi_{\max} = 32$, though for the $N = 128^3$ dataset it only barely out-performs the implementation of Section 5.2. However, for $N = 128^3$ and $\phi_{\max} = 1$, which has the largest number of nodes and spends the shortest amount of time computing AABBs for leaf nodes, the iterative methods perform poorly, suggesting that they are only beneficial for shallower trees with fewer iterations. Use of shared memory and coalescing are generally shown to be effective for this architecture.

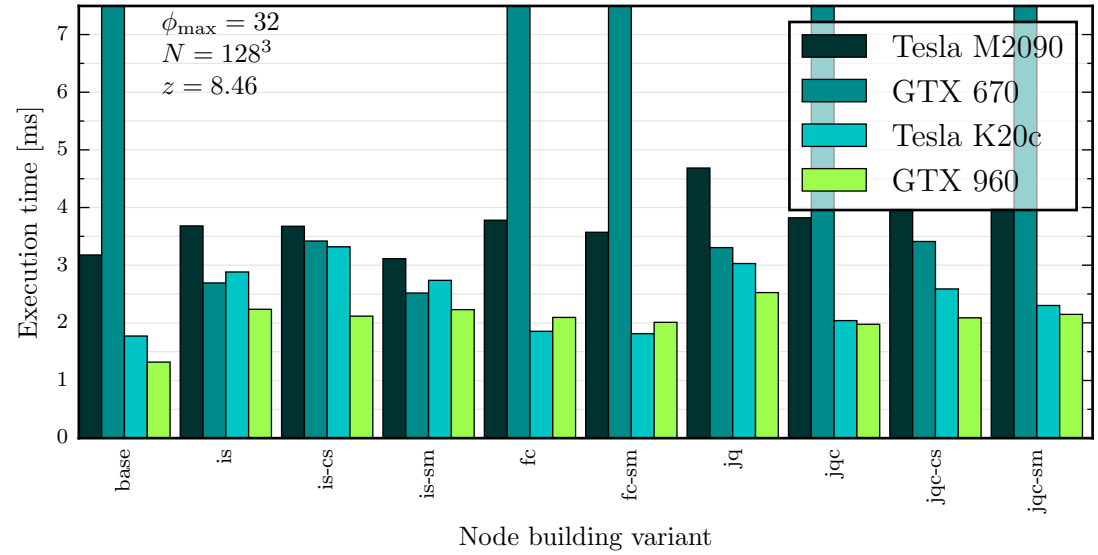
For the GTX670, the spuriously poor results on all implementations requiring a `__threadfence()` make it impossible to draw sound conclusions. However, the K20, which is also a Kepler-architecture device, favours the full climb implementation for all but the shallowest tree. This is most obvious for the $N = 256^3$ dataset, where we have both a large tree and a loop over many primitives to compute leaf AABBs. Increased use of shared memory is generally slightly beneficial, while coalescing is generally harmful.

Finally, the GTX 960 achieves highest performance with the implementation of Section 5.2.

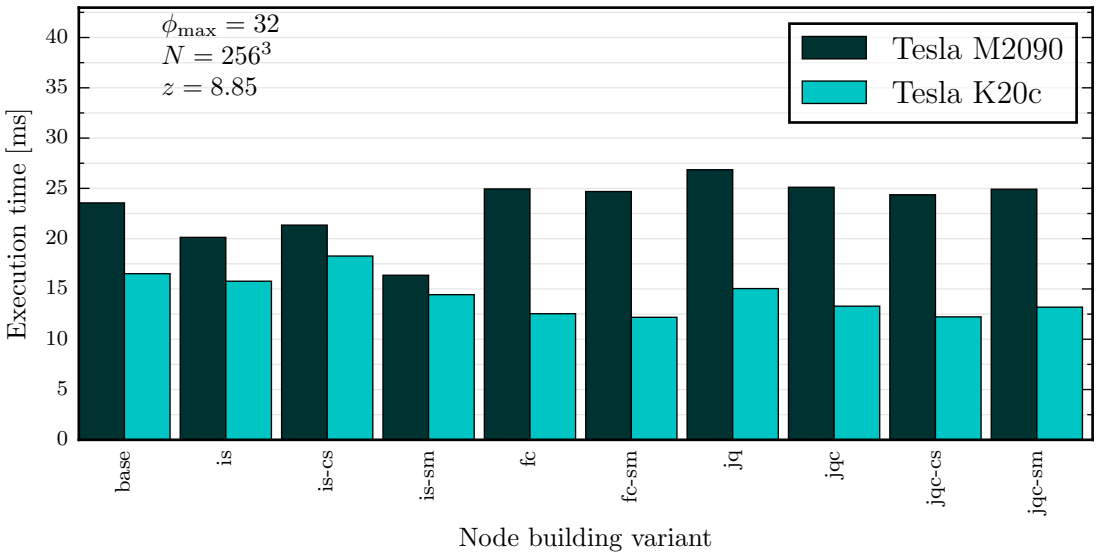
The overall picture, then, is a trend toward simple implementations, making use

Table 4.7. A summary of the node-build implementations presented in this section. The codes given here are used in Figure 4.12.

Code	Feature	Description
base	—	Unoptimized, as described in Section 5.2.
is	Iterative slices	The tree is built iteratively, in layers or slices, beginning with base nodes. No inter-block communication is required during an iteration. Each iteration ends by filling the work queue with base nodes at which to begin the next iteration.
jq	Join queues	Each thread climbs to the immediate parent of its starting node, propagating all necessary child data. Each fully-written parent node is written, contiguously, to a join queue (or work queue). All threads then exit. Nodes in the join queue are the starting point for the next iteration.
jqc	Join queues climb	Identical to join queues, except threads are permitted to climb some fixed number of levels, $n > 1$, before halting their climb and writing to the join queue.
fc	Full climb	Similar to "base", but split into two kernels. The first starts at all leaves, computes their AABBs, and propagates all necessary information to their parents. The second begins at these leaf parents, and then builds the entire hierarchy similarly to "base".
sm	Shared memory	An `atomicExch()` of the current thread ID to shared memory is used to distinguish the first and second threads to visit a node. Wherever possible, all indices and AABBs propagated up the tree are read from shared memory, rather than from global memory.
cs	Coalescing	After each child-to-parent climb, all threads in a block coordinate such that the smallest-possible number of warps are active for the next step (unless the number of active threads is ≤ 32).



(a)



(b)

Figure 4.12. Node-build time for the various implementations described in Table 4.7. Global memory operations are generally not included, unless they occur within a Thrust function, and with the exception of the various queue-compaction stages.

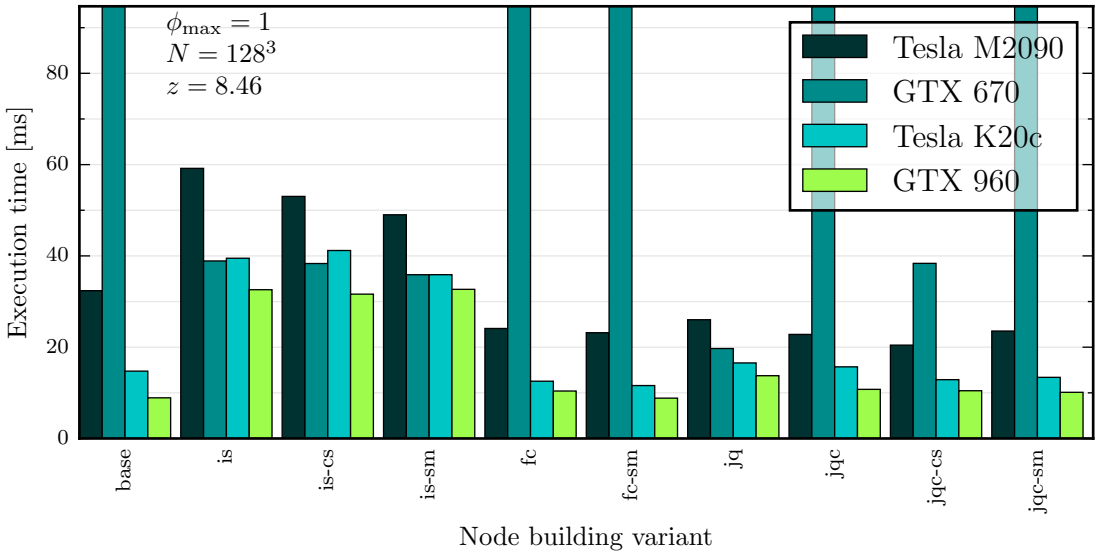


Figure 4.13. Node-build time, measured identically to the results in Figure 4.12, but with size-one wide leaves.

of shared memory where possible. Coalescing of threads to improve warp execution efficiency is helpful as often as it is harmful. The iterative scheme, which was effective when building the leaves themselves, performs very poorly for the largest and smallest number of nodes, but confusingly is somewhat effective for the intermediate $N = 256^3$ dataset. Join queues are often comparable to, but slightly worse than, the full climb implementation. Given its relatively good performance for all but the shallowest trees, the full climb implementation using shared memory (fc-sm) has been chosen as the final implementation in GRACE. This may be revisited in the future, given that the most recent architecture tested, Maxwell (GTX 960), appears to prefer the base implementation and a single kernel launch.

The combined performance of the 'fc-sm' node-build and 'lgss' leaf-build implementations, on a Tesla M2090, is given in Table 4.8, similarly as for Table 4.3 on page 89. (Note, however, that this node build implementation is actually a performance regression on this hardware.) For reference, recall that the highest performance for comparison codes was achieved by Karras (2012), with 200 primitives $\mathrm{ms}^{-1}\,\mathrm{GFLOPS}^{-1}$ and 1520 primitives ms^{-1} $\mathrm{GB\,s}^{-1}$. While falling short of these values, performance has close to doubled, and is now within a factor of ~ 2.6. Excluding the sort, as in Table 4.4, brings performance for the $N = 256^3$ dataset to within a factor of 2 of the results of Karras (2012). Comparisons with the same datasets are unlikely to be quite so favourable, as

Table 4.8. Comparison of the performance of unoptimized and optimized ALBVH implementations; c.f. Table 4.3. Morton key evaluation, sort-by-key and tree construction time are included; the Euclidean squared distance metric was used. Memory transfers and allocations within Thrust's `sort_by_key` function are the only ones included, and $\phi_{\max} = 32$. Performance is measured both in number of primitives per ms per unit throughput (GFLOPS) and number of primitives per ms per unit bandwidth ($\mathrm{GB\,s^{-1}}$). Results come from a Tesla M2090 device.

Dataset	Primitives/10^3	# primitives $\mathrm{ms^{-1}}$ / GFLOPS		# primitives $\mathrm{ms^{-1}}$ / $\mathrm{GB\,s^{-1}}$	
		Previous	Optimized	Previous	Optimized
$z \sim 11$, $N = 128^3$	2097	42.5	72.4	319	543
$z \sim 12$, $N = 256^3$	16 777	44.4	78.8	332	591

shown in Tables 4.3 and 4.4, and of course the results of Karras (2012) likely use leaf sizes of one, or at least substantially smaller than the 32 used here, and GRACE performs significantly worse in that case.

7 Ray Tracing Implementation

7.1 Casting rays

Generally, rays may be generated by an external method and passed to the traversal algorithm. However, there are some ray distributions common to radiative transfer problems for which I have added ray-generating methods to GRACE. The rays themselves are represented by eight floating-point values. Three specify an origin, O, and three more a normalized direction vector, $\hat{d}$. The final two values specify the start and end points of the ray (a line segment) as measured along $\hat{d}$ from O. Such a layout is often convenient for ray-primitive intersections tests, and may be realized as two `float4`-like values, allowing for efficient access.

Before continuing, the existence and applicability of HEALPix (Górski et al., 2005) should be noted. The HEALPix algorithm is able to divide a spherical surface into a semi-arbitrary number (12×4^n for integer n) of equal-area pixels. It also provides the normal vectors for these pixels, which are readily interpreted as the unit vectors of a set of isotropically-distributed rays. A key advantage is that, because pixels can be refined and merged, the resolution of the resulting rays is similarly adaptive (e.g. Wise and Abel, 2011; Greif, 2014, to name but a few examples). While such usage is clearly of

benefit, it typically requires some level of book-keeping, with ray refinement specific to the problem at hand, and often a function of the local density field. My position is, therefore, that it is best implemented by users where needed; as previously stated, GRACE will accept any arbitrary set of rays in addition to those it is able to generate.

Most-useful for radiative transfer are isotropic ray distributions. The implementation here makes use of the random number generator (RNG) library provided as part of the CUDA toolkit, cuRAND.[14]

An isotropic distribution of normalized direction vectors with a common origin is equivalent to a distribution of uniformly-random points on the surface of the unit sphere. One method of drawing points distributed in this manner is to first draw three independent, normally-distributed numbers, x, y and z, and to then compute

$$\hat{\mathbf{r}} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} (x, y, z). \tag{4.9}$$

This is justified in Appendix B. The cuRAND library provides on-GPU functions for several RNGs, and for the above `curand_normal()` is used.

In some cases, it may be desirable to generate an isotropic distribution within some solid angle, $\Omega \leq 4\pi$, centred about some arbitrary direction $\hat{d}$. For the case that $\hat{d}$ points along the positive z-axis, this can be achieved by first generating two uniform random variables,

$$\begin{aligned} \beta &\in (0, 2\pi], \\ z &\in (1 - \Omega/2\pi, 1], \end{aligned} \tag{4.10}$$

realized here via a call to `curand_uniform()` and appropriate scaling, and then computing the direction $\hat{\mathbf{r}} = (x, y, z)$, where

$$\begin{aligned} x &= \sqrt{1 - z^2} \cos\beta, \\ y &= \sqrt{1 - z^2} \sin\beta, \end{aligned} \tag{4.11}$$

which is clearly normalized. This is derived in Appendix B. It then serves simply to apply the rotation matrix M, where $\hat{d} = M\hat{z}$, to all such generated vectors. This is again given in Appendix B.

The cost of initializing RNG states is reduced by executing, a single time, a dedicated

[14] For details, see `http://docs.nvidia.com/cuda/curand/`

initialization kernel and saving the resulting states to global memory for later (re-)use. Further, the total number of states produced is only a factor of a few greater than the number of threads which may be *simultaneously* executing instructions on the hardware. The aim here is to require as few RNG states as possible, while still ensuring that execution units are not starved of work.

In the likely case that $N_{\text{rays}} > N_{\text{states}}$, each ray-generating thread calls `curand_[normal|uniform](&rng_state)` multiple times, until it has generated its quota of rays.

Each RNG receives the same seed parameter, ensuring an identical set of rays can be generated on multiple invocations of the program. Following the documentation, correlations between generators are avoided by providing each a different sequence identifier, set equal to its thread's ID.

Before verifying isotropy, we will first examine the performance of the various pseudo-random generators available in cuRAND. The time taken to initialize states (the number of which varies by device) and generate 3×10^6 normally distributed `float` values is given in Table 4.9 for three generators: Philox (Salmon et al., 2011), XORWOW (Marsaglia, 2003) and MRG32 (L'Ecuyer, 1999; L'Ecuyer et al., 2002). For each set of three values x, y and z, the kernel computes $t = x * y + z$ as a single FMA instruction and increments a per-state counter if the least-significant bit of t is set.[15] The counter and the (updated) state are saved to global memory only after a state has generated all its $\sim 3 \times 10^6 / N_{\text{states}}$ values. No other memory operations are included.

I have opted to use the XORWOW generator by default in GRACE (it is also the CUDA default). A typical use-case for GRACE will initialize states only once, but use them to generate rays many times, so the smallest T_{gen} is preferable.

To confirm that the implementation, which generates a ray distribution according to Eq. (4.9), can be considered sufficiently isotropic, several statistical tests are employed.[16]

The first is Ripley's K-function, a test of spatial homogeneity (see e.g. Dixon, 2002). It is defined as

$$K(s) = \frac{1}{\lambda}\mathbb{E}[N(s)], \tag{4.12}$$

[15] It was found that x, y and z must be used in some manner to obtain meaningful results; if unused, the compiler is free to optimize them out, retaining only the update to the underlying state. This is particularly noticeable for Philox, where the state transition is a very simple operation.

[16] The usefulness of these tests is somewhat dubious. At their core, they simply compare results from one random number generator to another; here, the above cuRAND to that of `numpy`. Nonetheless, bugs in the implementation can be — and, in fact, were — detected.

Table 4.9. Comparison of the performance of three RNG generators available in cuRAND. In all cases, 3×10^6 pseudorandom numbers were generated. Highlighted in bold are the shortest initialization and generating times for each device.

Device	Generator					
	T_{init} (ms)			T_{gen} (ms)		
	Philox	XORWOW	MRG32	Philox	XORWOW	MRG32
M2090	**0.10**	2.1	6.6	0.77	**0.61**	0.79
GTX 670	**0.041**	2.0	7.6	0.49	**0.30**	1.5
K20	**0.067**	4.3	4.7	0.61	**0.45**	0.58
GTX 960	**0.037**	1.5	13	0.50	**0.38**	1.8

where $N(s)$ is the number of other points within a distance s of a randomly chosen point, and λ is the number of points per unit area. For a collection of n points with positions $\mathbf{r}_i$, the unbiased estimator is

$$\hat{K}(s) = \frac{1}{\lambda(n-1)} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i \neq j}}^{n} I(|\mathbf{r}_i - \mathbf{r}_j| < s), \tag{4.13}$$

where I evaluates to 1 when its argument is true, and to 0 when it is false. To adapt the estimator for spherical geometry (see Robeson et al., 2014, for a derivation), we replace $|\mathbf{r}_i - \mathbf{r}_j|$ with $|\mathbf{r}_i - \mathbf{r}_j|_{\mathbb{S}}$, defining the great-circle distance between the two points on the unit sphere. The scale s then also refers to a great-circle distance, and the value under complete spatial randomness (CSR) is the area of the spherical cap on the unit sphere, $2\pi[1 - \cos(s)]$.

The second is the (modified) Rayleigh statistic, which tests the null hypothesis that there is no mean direction to the distribution (see Fisher et al., 1987, Chapter 5). For n points on the unit sphere, with co-ordinates $\mathbf{r}_i = (x_i, y_i, z_i)$, it is defined as

$$\mathcal{W}^2 = \frac{3}{n} \left\{ \left(\sum_{i=1}^{n} x_i \right)^2 + \left(\sum_{i=1}^{n} y_i \right)^2 + \left(\sum_{i=1}^{n} z_i \right)^2 \right\}. \tag{4.14}$$

The null (isotropic) distribution for $\mathcal{W}$ as $n \to \infty$ is χ^2_3; such an interpretation for $\mathcal{W}$ is appropriate for $n \gtrsim 10$ (Diggle et al., 1985). While useful, $\mathcal{W}$ is not sensitive to distributions which are non-isotropic but symmetric with respect to the centre of the sphere; similarly, it is also most-sensitive to unimodal distributions.

To test the null hypothesis of an isotropic distribution against the alternative of some preferred direction(s), Fisher et al. (1987) again offer a solution, in the form of Beran's statistic, $\mathcal{A}_n$, and Gine's statistic, $\mathcal{G}_n$,

$$\mathcal{A}_n = n - \left(\frac{4}{n\pi}\right) \sum_{i<j\leq n} \theta_{ij}, \tag{4.15a}$$

$$\mathcal{G}_n = \frac{n}{2} - \left(\frac{4}{n\pi}\right) \sum_{i<j\leq n} \sin(\theta_{ij}), \tag{4.15b}$$

where the sum is over all *distinct* pairs of points, and θ_{ij} is the angle between points i and j, defined by $\hat{\mathbf{r}}_i \cdot \hat{\mathbf{r}}_j \equiv \cos(\theta_{ij})$, with $\theta_{ij} \in [0, \pi]$. $\mathcal{A}_n$ tests the null hypothesis of a uniform distribution against alternative models which are not symmetric with respect to the centre of the sphere, while $\mathcal{G}_n$ tests against alternative models which are.

Since our goal is to, in some sense, 'confirm' isotropy, we must test the null hypothesis that the ray distribution is *not isotropic*, subject to some *a priori* definition of 'not isotropic'. For normally-distributed statistics, the two one-sided test (TOST) (see e.g. Schuirmann, 1987) serves this purpose, and is outlined in Appendix D. In essence, a confidence interval for the difference in the mean of a test sample and reference sample is computed at some chosen confidence $1-\alpha$. If this is fully contained within a region of equivalence, the null hypothesis of *non*equivalence is rejected, in favour of the alternative hypothesis of equivalence.

For non-normal data, instead a procedure based on the non-parametric Mann-Whitney U-statistic is used (Wellek, 1996), also outlined in Appendix D. We generate a test statistic and compare it to a critical value. The test statistic is a function of the two distributions to be compared and the region of equivalence, while the critical value depends on the significance level, α.

I have selected the relatively conservative value $\alpha = 0.005$.

For testing, 200 samples, each containing 9600 rays, are generated by the GRACE implementation. The reference (assumed isotropic, or CSR in the terminology of the K-function) samples are generated using the Python package NumPy[17]. In an effort to prevent identical errors occurring in both implementations, reference samples are generated via rejection sampling. 2000 samples of 9600 directions are used for the reference.

Visual inspection of $\hat{K}$-statistics shows a good fit to a (maximum-likelihood) estimate

[17] `http://www.numpy.org/`

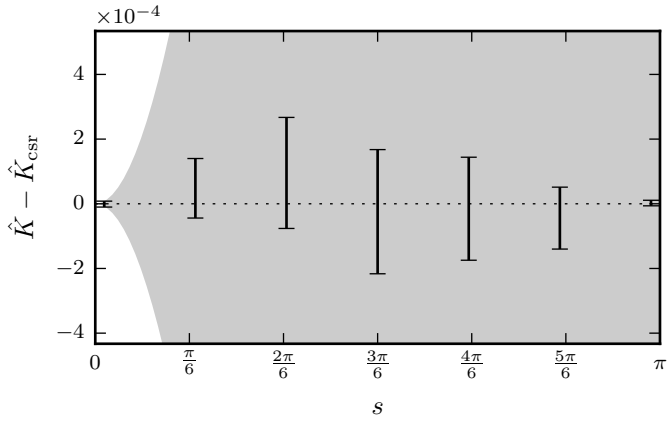


Figure 4.14. Ripley's K function at various on-sphere scales. $\hat{K}$ is the value computed from the GRACE implementation, and $\hat{K}_{\mathrm{csr}}$ from the assumed-isotropic samples. Vertical bars denote the 99.5% confidence intervals for $\hat{K} - \hat{K}_{\mathrm{csr}}$. Shaded is the region of equivalence; it is bounded bounded by $\hat{K} = (1 \pm 0.001\hat{K}_{\mathrm{csr}})$.

of Gaussian parameters at all but the smallest scales. Figure 4.14 gives a graphical presentation of the TOST at a selection of scales in $(0, \pi)$. A confidence level of $1 - \alpha$ applies independently to each tested scale. For the equivalence region, it is (arbitrarily, but quantitatively) supposed that an over-counting or under-counting of 0.1% at each value of i in Eq. (4.13) can be considered practically equivalent.

Only at the very smallest scale ($s = 0.05$) is there any cause for concern: the result is inconclusive — rays may be clustered, dispersed, or consistent with the reference sample. Note that here we expect only $O(1)$ other points to lie with $s = 0.05$ of any given point. Given the outcomes at other scales, no further investigation is attempted. Similar results are achieved with both the Philox and MRG32 RNGs.

The Rayleigh, $\mathcal{A}_n$ and $\mathcal{G}_n$ statistics do not present an obvious algebraic manipulation for computing an equivalence region. Instead, 2000 samples of deliberately-biased direction vectors are produced, again with 9600 directions per sample. Bias is quantified in terms of a bias fraction, $f \in [0, 1/2]$, and a number of preferred directions, or modes, m. The preferred directions are themselves specified by a primary direction $\hat{d}_i$ and an opening angle $\theta_i \in (0, \pi]$ (i.e. they are spherical cones).

For a non-symmetric biased sample of n directions (testing $\mathcal{W}$ and $\mathcal{A}_n$) with $m = 1$, an excess of nf direction vectors point into the preferred-direction cone relative to that expected from a uniform random sample of n directions. For $m > 1$, the number of biased direction vectors is identical to the $m = 1$ case. Each of the bias vectors is then randomly assigned to one of the m cones.

Table 4.10. The test statistics for a noninferiority test of GRACE rays against an assumed-isotropic reference sample. Values greater than 2.576 reject the null hypothesis of inferiority at the 99.5% confidence level.

Statistic	Noninferiority test value	Result
$\mathcal{W}$	5.605	Reject null
$\mathcal{A}_n$	6.027	Reject null
$\mathcal{G}_n$	3.969	Reject null

For a symmetric biased sample of n directions (testing $\mathcal{G}_n$), again there is an excess of nf total bias vectors, but exactly half of these are the point reflections of the other half.

A value of $f = 0.5\%$ was chosen, hence $nf = 48$. An opening angle of $\pi/2$ is used for all cones, with $m = 1$ in all cases.[18] The critical value is 2.576.[19] For all of the Rayleigh, $\mathcal{A}_n$ and $\mathcal{G}_n$ statistics, larger values are inferior (less isotropic). The results are given in Table 4.10.

The samples generated by GRACE reject the null hypothesis of inferiority for all statistics. This result also holds for both the Philox and MRG32 RNGs.

7.2 Computing the SPH integral

An approximation for the line integral of an SPH density field was given in Chapter 2, Section 4.3 as Eq. (2.44), reproduced below,

$$N_{\mathrm{cd}} \approx \sum_{i=1}^{N} \int_{l_i^{\mathrm{in}}}^{l_i^{\mathrm{out}}} m_i \, W[r(l),\, h_i] \, \mathrm{d}l, \tag{4.16}$$

where the sum is over all N intersected particles for a given ray, m_i is the mass of each such particle, h_i the smoothing radius, $r(l)$ is the Euclidean distance from the point a distance l along the ray to the particle centre, W is the SPH kernel, and l_i^{in} and l_i^{out} are, respectively, the entry and exit distances along the ray. This is illustrated in Figure 4.15.

We thus require a method of computing the integral in Eq. (4.16). Henceforth the mass component m_i will be dropped, as it is a per-particle constant; the method is hence

[18] For a fixed value of f, some empirical investigation suggests that all statistics are most sensitive at $m = 1$.

[19] $1 - \Phi(2.576) \approx 0.005$, where Φ is the cumulative standard normal distribution.

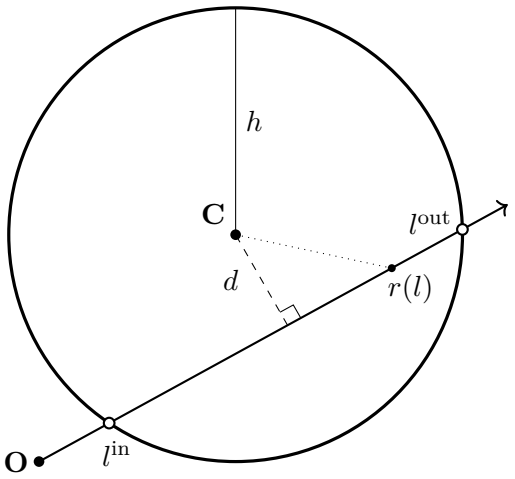


Figure 4.15. An illustration of the terms involved when computing the per-particle SPH integral along a ray the ray-sphere intersection test. The sphere is of radius h, with centre **C**. l^{in} and l^{out} denote the distances along the ray at which it enters and exits the sphere, respectively; d the distance of closest approach; and $r(l)$ the distance from **C** a distance l along the ray.

applicable to other scalar properties of an SPH field. The calculation depends on the choice of the smoothing function W, and here we adopt the GADGET-2 kernel, $W(r,\, h)$ (Springel, 2005). This was given in Chapter 2, Section 3.3 (page 22) as Eq. (2.31), but is repeated here in terms of the parameter $x(l) \equiv \dfrac{r(l)}{h}$,

$$\frac{8}{\pi h^3}\hat{W}(x) \equiv W(r,\, h) \equiv \frac{8}{\pi h^3}\begin{cases} 1 - 6x^2 + 6x^3 & 0 \leq x \leq \dfrac{1}{2} \\ 2(1-x)^3 & \dfrac{1}{2} < x \leq 1 \\ 0 & x > 1 \end{cases}. \tag{4.17}$$

If we take that $l^{\text{out}} = -l^{\text{in}} \equiv l$, then from Eq. (4.17), it is clear that the integral may be rewritten as

$$\mathcal{I} = \frac{16}{\pi h^3}\int_0^l \mathrm{d}l'\, \hat{W}[\, x(l')\,]. \tag{4.18}$$

From Figure 4.15, we then have that

$$(hx)^2 = d^2 + l^2, \tag{4.19}$$

and so

$$\mathrm{d}l = \frac{x}{\sqrt{x^2 - b^2}}\,\mathrm{d}x, \tag{4.20}$$

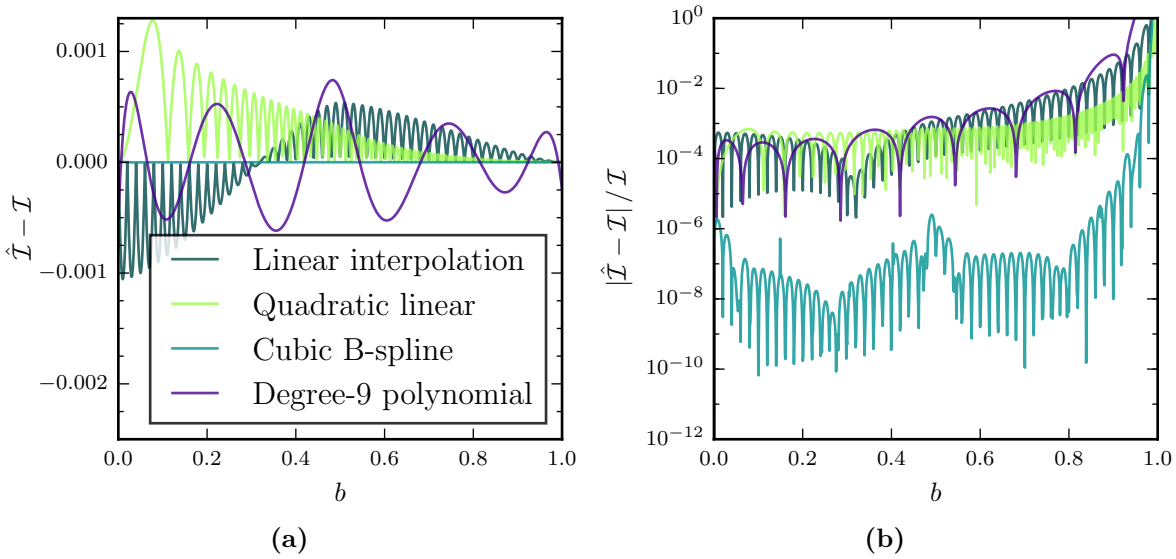


Figure 4.16. The error in several approximations, $\hat{\mathcal{I}}$, of the SPH kernel line integral, $\mathcal{I}$. **a)** the absolute error in each approximation; **b)** the fractional error.

where $b \equiv \dfrac{d^2}{h^2}$ is the ray-particle impact parameter.

Finally,

$$\mathcal{I} = \frac{16}{\pi h^3} \int_b^1 \mathrm{d}x \, \frac{x \, \hat{W}(x)}{\sqrt{x^2 - b^2}}, \tag{4.21}$$

which is conveniently parameterized; in particular, we may pre-compute the integral for various values of b and scale the result with h at runtime.

An analytic solution to Eq. (4.21) does exist, but contains a large number of terms (including the computationally expensive natural log and square root). Approximate solutions are therefore sought. Linear interpolation from tabulated values, B-spline approximation and polynomial fitting are obvious candidates. To assess these options, Eq. (4.21) is computed, numerically, for 50 evenly-spaced values of b in [0, 1] using the `romberg` integrator from the Python package SciPy.[20] (While Eq. (4.21) is useful to demonstrate the parameterization over b, in practice the denominator's singularity at $x = b$ is problematic, and the integral is more easily computed over l.) The error and fractional error in each of these three methods is shown in Figure 4.16.

Fitting of two functional forms,

$$f_1(b) \equiv c_1 \cos(c_2 b) \exp(-c_3 b^2), \tag{4.22}$$

$$f_2(b) \equiv \frac{c_1}{1 + \exp\left[-c_2(b - c_3)\right]}, \tag{4.23}$$

[20] `https://www.scipy.org/`

was also attempted, but both resulted in larger errors than the above linear interpolation for (almost) all values of b.

A degree-9 polynomial produces errors roughly comparable to that of linear interpolation, but increases traversal time by a factor of ~ 1.5. While the spline produces, by some margin, the closest approximation, I consider the simpler linear interpolation to be sufficiently accurate. It does, however, require computation of a square root, as the ray-sphere intersection routine produces the value b^2. To avoid this, it is possible to perform linear interpolation in b^2, the result of which is given as 'Quadratic linear' in Figure 4.16. Evenly-spaced values of b^2 lead to undersampling at low values of b; to compensate, 80 values are tabulated, rather than 50. This reduces the execution time by approximately 5%.

To assess the accuracy of this somewhat *ad hoc* scheme, the volume integral of a 128^3 particle, $z \sim 20$ SPH field is estimated as a sum of 262 144 line integrals, multiplied by a per-line effective cross-sectional area (see Section 8 for further details). For the b^2, 80-value interpolation, $\hat{V}_{80}$, and the b, 50-value interpolation, $\hat{V}_{50}$, we find a fractional delta of $|\hat{V}_{80} - \hat{V}_{50}| \,/\, \hat{V}_{50} = 4.0 \times 10^{-5}$.

The texture cache on CUDA GPUs is able to perform interpolation in hardware. This acceleration does entail a reduction in accuracy: a low-precision format is used for the interpolation, such that there are only 255 possible interpolated values between adjacent tabulated entries. It is likely that this would provide sufficient resolution with a table of 50 values. However, no performance gain was observed on implementing in-hardware interpolation.

Moving the tabulated values to shared memory resulted in a $O(1\%)$ reduction in runtime, depending on the device. Fermi devices in particular show little gain, suggesting that the lookup table was generally available in L1 cache.

7.3 BVH traversal

Naïve traversal

BVH traversal is often expressed recursively: begin at the root node with `traverse(root, ray)`; the function checks both child nodes for intersection, and calls `traverse(child, ray)` for each intersected child. This is efficiently implemented in GRACE using a *stack*, a container where the most-recently inserted (`stack.push()`) item is the first to be removed (`stack.pop()`).

Listing 4.8. Stack-based traversal.

```
1  stack.push(root);
2  while (!stack.empty())
3  {
4    Node node = stack.pop();
5
6    while (node.is_inner())
7    {
8      ResultType hit = intersect(node, ray);
9
10     if (hit.left)
11       stack.push(node.left);
12     if (hit.right)
13       stack.push(node.right);
14   }
15
16   while (node.is_leaf())
17     intersect(node.primitives, ray);
18 }
```

The basic procedure is outlined in Listing 4.8, and results in a *depth-first* traversal order. Each thread loads a single ray, and performs its own, independent traversal. The stack is realized as a fixed-size, thread-local `int` array, containing the indices of nodes which are in the stack. The `push` and `pop` operations then essentially reduce to simple pointer increments, decrements and dereferences.

Note that either of the `while` statements on Lines 6 and 16 could be replaced with `if` statements. In GRACE, I have opted for `while-while`, following the findings of Aila and Laine (2009), having confirmed that it typically results in a reduction of $O(10\%)$ for traversal times relative to the `if-if` alternative.

Increasing the number of particles in each leaf node, $\phi_{\max}$, from 1 to the approximately-optimal value of 32 consistently halves the runtime of the trace kernels. While the exact value of $\phi_{\max}$ does not appear to be particularly important, having it be $O(10)$ certainly is. A sharp reduction in traversal time is seen as $\phi_{\max}$ is increased from 1 to ~ 20. A much gentler increase in runtime is then seen once $\phi_{\max}$ passes its optimal value. This general trend holds over most SPH datasets, as shown under 'No packet' in Figure 4.17, page 124, and there are no significant deviations from this behaviour.

In the following sections, various optimizations for the traversal procedure are investigated, in the context of finding *all* ray-particle intersections for SPH datasets.

Optimizations are listed approximately in the order they were implemented. Where performance improvements are quoted, they are relative to previous results, not relative to the base implementation just described.

Memory layout of nodes

Just as for SPH particles (see Section 2), both a structure-of-arrays (SoA) and an array-of-structures (AoS) in-memory layout for the nodes should be considered (see Listings 4.1 and 4.2). It is found that, for tree construction on SPH datasets, an SoA can reduce the time taken by up to a third. However, for an (artificial) fully-balanced tree — where all root node to leaf node paths pass through the same number of inner nodes — performance may actually be reduced. Further, the traversal performance — which typically dominates the overall runtime — is markedly worse for SoA, increasing by up to 40%.

That tracing strongly favours the AoS data structure is not unexpected. Threads will be accessing many different nodes simultaneously, and such a GPU-hostile access pattern is best combated by ensuring the maximum utility of each load instruction. That is, we should issue the widest-possible vector load instruction, whilst loading only data which is actually required. AoS-packing of both child indices and all six AABB co-ordinates into a `struct` works toward this goal. It can be improved upon further by placing both child AABBs within a node, rather than its own AABB. Each node then contains two `int` values (its child indices) and 12 `float` values (its two child AABBs), for a total of 56 bytes. The BVH construction procedure (see Section 5) typically requires a further two `int` values, bringing each node to a convenient total size of 64 bytes. Thus, when tracing, each thread requests a contiguous 64-byte section of memory, a significant chunk of a 128-byte L1 cache-line, and double the 32-byte L2 cache-line. All 64-bytes are packed into four `float4`-like structures, and thus require only four vector load instructions.

A further $O(10\%)$ reduction in traversal time is obtained from this two-AABB packing, relative to the basic AoS, but tree-construction is not significantly affected.

Packet traversal

The memory access pattern may be further improved by forcing consecutive groups of threads to follow identical paths through the BVH, even when they might not otherwise.

A warp is a convenient such grouping, and this form of packet traversal was first suggested for GPUs by Gunther et al. (2007). (Note that, in the literature, a 'packet' typically refers to a grouping of rays which are traced simultaneously *by a single thread*. For example, one may compute a bounding cone or bounding frustum for the rays, and test this for intersection with the BVH; CPU SIMD instructions also allow for multiple rays to be tested against an AABB in fewer instructions than a simple `for` loop.)

The basic procedure is identical to Listing 4.8, except that all threads within a warp share the same stack. The two `if (condition)` statements at Lines 10 and 12 are modified to `if (__any(condition))`, which evaluates to true if `condition` evaluates to true for any thread(s) in the warp. Hence all threads in a warp always request the same contiguous 64-bytes when reading a node. Technically, warp divergence is also completely eliminated; in reality, however, threads may do unnecessary work.

To minimize the overhead of unnecessary ray-node intersection tests, rays within a warp should follow similar paths through the BVH under independent traversal. For randomly distributed rays emanating from a common origin, this is achieved by first computing the 30-bit Morton key (recall Chapter 3, Section 4.2) of each ray's normalized direction vector, and then sorting rays by their key values. Morton key sorting can easily be extended to arbitrary ray distributions by concatenating the Morton keys of the origin and direction vectors. Where rays are generated for the purposes of image production, they are likely to already be in an acceptable order.

Packet traversal consistently reduces the runtime of the traversal kernels by at least 25% for SPH datasets, and by up to 60% for some datasets on the Fermi architecture, as shown by the '32-packet' and 'No packet' lines in Figure 4.17. Note, however, that it is more effective for larger values of N_{ngb} and smaller values of N. This is further discussed in Section 9.5.

Initially, the shared stack was implemented as a true shared stack — i.e. one per warp — in shared memory; however, it was found that a per-thread stack in thread-local memory is equally effective.

Figure 4.17 also shows the contribution of node traversal, ray-primitive intersection and intersected-primitive processing; here, the processing entails cumulating the SPH integral along each ray. These measurements were made by guarding relevant code with `if` conditionals designed to be always true, or always false, in a manner that would not be visible to the compiler — for example, that the number of primitives in a leaf is

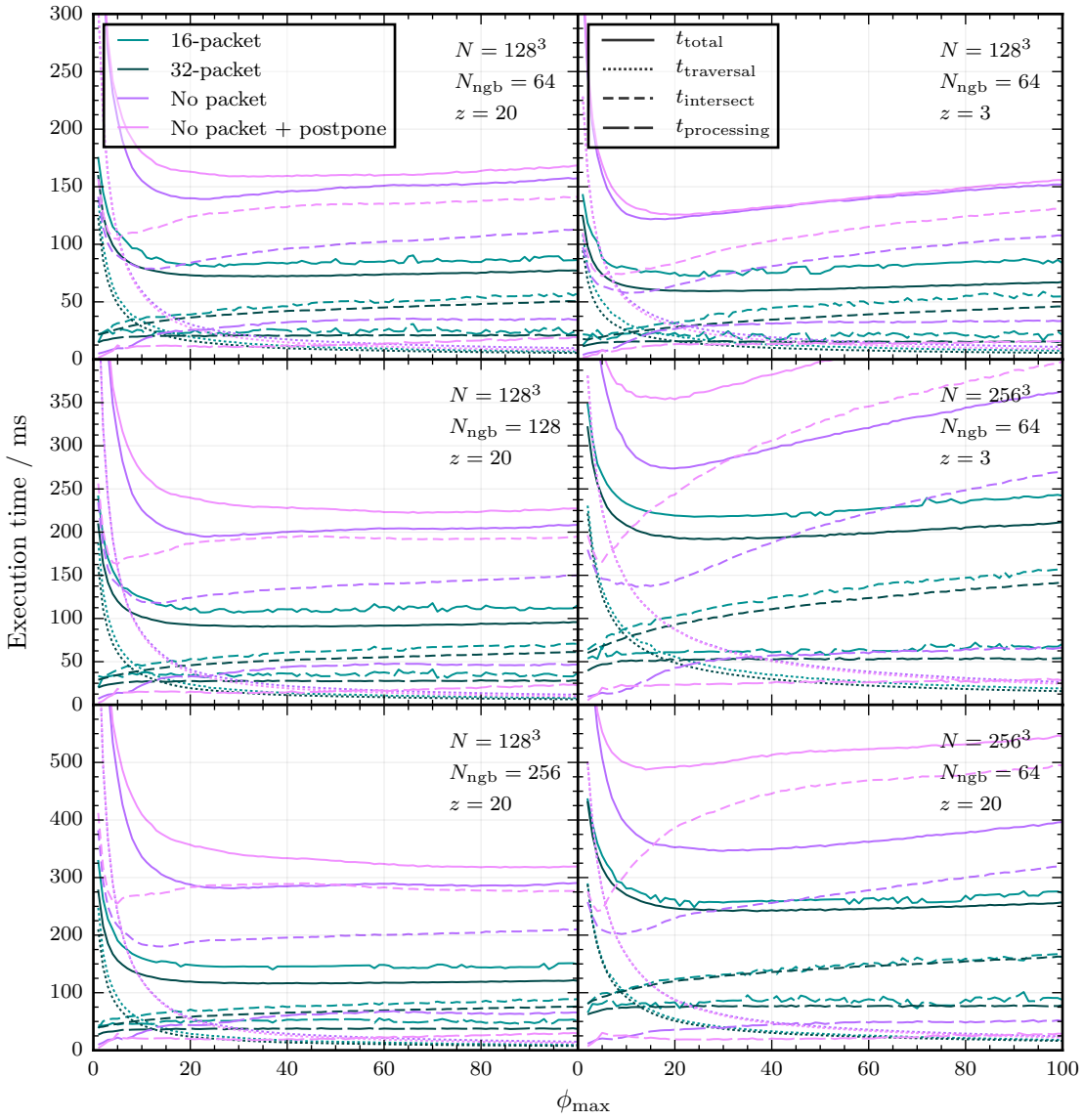


Figure 4.17. The traversal performance for packet tracing, independent-ray tracing, and independent-ray tracing with leaf traversals postponed. In all cases, 38 400 rays were traced from a common origin, all exiting the simulation volume, and the total SPH-particle integral cumulated along each ray. N refers to the number of particles, N_{ngb} the number of near-neighbour particles (which has been artificially increased from 64), and z the redshift. $t_{\text{traversal}}$ is the time due to only the BVH traversal, $t_{\text{intersect}}$ the ray-sphere intersection tests, and $t_{\text{processing}}$ the processing of intersected particles. All results were obtained from a Tesla M2090 (Fermi) device.

positive, or negative, respectively. In a first run of the kernel, all conditions evaluate to true; the second ensures that no intersected particles are processed; the third that no ray-particle intersection tests are made. If we assume that

$$t_{\text{total}} = t_{\text{traversal}} + t_{\text{intersect}} + t_{\text{processing}},$$

the values on the right-hand side are then trivial to compute. When switching from an always-true to an always-false guard, no extra data is needed, and the number of instructions is the same. Comparison of t_{total} to a variant with no such guards showed a $\lesssim 1\%$ increase in runtime. Finally, manual inspection of the assembly for the compiled kernels, using `cuobjdump`, verified that the compiler made no changes to invalidate the results (that memory loads may be moved, relative to the conditionals, was of particular concern).

It is an interesting result of this analysis that the gains for packet tracing primarily come from improved ray-primitive intersection tests. This strongly suggests that this aspect of traversal is heavily dependent on bandwidth, benefiting from the coalesced accesses that packet tracing allows. In both relative and absolute terms, node traversal does not see such large gains, but certainly they are still significant at lower values of $\phi_{\max}$, again pointing to bandwidth limitations.

Figure 4.17 also shows results for 16-wide packets. These were implemented just as 32-wide packets, but all packet-wide `__any()` instructions were modified to `MASK & __ballot()`, where `MASK` was `0xffff` for the lowest 16 threads in the warp, and `0xffff0000` for the highest 16. This requires the mask to be set at runtime, incurring some additional overhead.

Postponed traversal

Aila and Laine (2009) present a method of postponed (or speculative) traversal, intended to reduce the warp divergence of independent-ray traversal.

If a ray finds a leaf to be intersected, it adds the leaf to a queue. The thread is then free to continue intersecting other nodes, rather than idling until the other threads in the warp also move to leaf traversal (warp divergence). When all threads have at least one item in the queue, the warp moves to leaf traversal. If a thread's leaf queue is full, it must still idle until the warp moves to leaf traversal. Performance was highest when

the leaf queue had a size of one (being implemented as a single `postponed_leaf_index` variable). However, as shown in Figure 4.17, overall performance is consistently worse than the standard independent-ray traversal. This is entirely a result of increased intersection times; BVH traversal is largely unaffected, and processing time is actually reduced. This further suggests that the success of packet tracing is due to its effect on the memory access pattern — increasing warp efficiency has reduced performance! This is further discussed in Section 9.5.

That processing is the only stage to see an improvement suggests it is not limited by the available bandwidth. The significant reduction in processing times relative to packet traversal, in particular, is interesting. A reasonable explanation is that, when searching an intersected leaf node, few rays in a packet actually intersect any of the leaf's primitives. Indeed, this is found to be the case: for an intersected leaf, and on average, only ~ 5 rays within a packet go on to produce particle intersections. This highlights an inefficiency of packet traversal at the scale of leaf nodes. A hybrid implementation, combining the favourable traversal and intersection tests of packets with the fine-grained primitive-processing of a leaf-postponing implementation is worthy of investigation. However, given that even the rather substantial particle processing done here accounts for only $\lesssim 20\%$ of the total tracing time, I have not pursued this further.

Stackless traversal

It is possible to achieve a depth-first traversal without the use of a stack. This also solves a practical issue inherent to the use of a stack — its size must be specified and, when using thread-local or shared memory, it must be specified at compile time. (Nonetheless, a stack size of 64 elements has proved more than sufficient for SPH datasets up to at least 256^3 particles.)

If the left child of a node is always the first to be traversed (as is the case in Listing 4.8), the order in which nodes are visited is well determined. This is shown in Figure 4.18. A depth-first node-ordering — also shown in Figure 4.18 — then allows for a simple traversal algorithm. If a node, p, is intersected, the next node in the traversal is its left child, always located at $p + 1$. If it is missed, the entire sub-tree below p is skipped. Now, suppose p contains n_L leaf nodes and, therefore, $n_L - 1$ inner nodes. The total number of skipped nodes is $2n_L - 1$, and the next node in the traversal is located

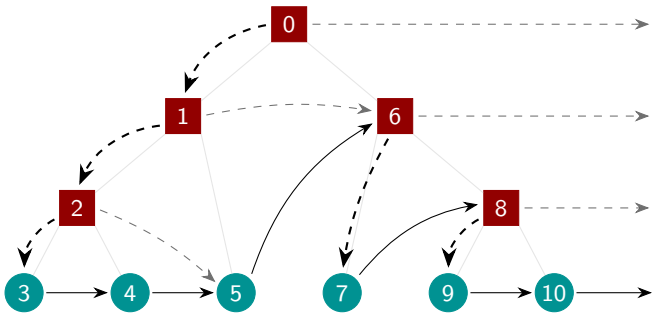


Figure 4.18. The depth-first traversal of a ray through a BVH. Solid lines denote traversal paths which are always taken after a leaf has been processed. Dark dashed lines are taken if a node is intersected. Light dashed lines are taken if a node is missed. Paths which do not point to another node imply that the traversal is complete.

at $p + 2n_L - 1$. If this index is greater than the index of the final node, the traversal is complete. (A leaf node is considered to have $n_L = 1$.)

In practice, obtaining such a node ordering may be expensive in terms of both computation and memory. For an arbitrary BVH, one may perform a single depth-first traversal using a stack, and populate nodes with jump-to indices corresponding to the various arrows in Figure 4.18. Here, I have implemented stackless traversal for the hierarchy generated by the method of Karras (2012). In short, it produces two arrays, one for nodes and one for leaves. A left-child inner node has the same index as its right-most leaf, and a right-child inner node has the same index as its left-most leaf. The root node has index 0. Child indices are stored in each node, immediately producing all on-hit arrows in Figure 4.18, while on-miss indices may be computed efficiently in parallel: a thread at a node p proceeds as follows

- Set the on-miss index of the left child equal to the index of the right child.
- Examine node $p + 1$. If it is an ancestor, set the on-miss index of the right child equal to the *leaf node* $p + 1$, otherwise to the *inner node* $p + 1$.

(If the set of leaves contained in node p are also contained in $p + 1$, then $p + 1$ is an ancestor of p.) This results in a negligible ($< 1\,\text{ms}$) increase in BVH construction time.

An unfortunate consequence of this procedure is that the two-AABBs node layout described earlier is not entirely compatible. It is possible that a jump-to index will correspond to a node whose AABB has already been tested, and for which no intersection was found. Hence the stackless implementation may do unnecessary work. For this reason, the node `struct` is here reverted to contain its own AABB. Comparing stackless and with-stack implementations, both utilizing this sub-optimal node structure, does

not identify a clear winner; neither implementation consistently out-performs the other. It would therefore appear that operations on the stack rarely have a significant impact on traversal performance. And, of course, both perform worse than the two-AABB node traversal with stack.

In light of the above, and the potential cost of computing jump-to indices for other BVH layouts like ALBVH, I have opted not to make use of a stackless traversal.

Improving coalescing on memory stores

For some use cases, data must be stored for all ray-primitive intersections — for example, the SPH integral, the particle index, and the distance to the intersection. This is the case for TARANIS (see Chapter 5, Section 3), so the discussion here assumes exactly these output requirements. Even for a small, 128^3 particle dataset, at $N_{\text{ngb}} = 64$ each ray intersects $O(10^3)$ particles, which for the above example equates to $\sim$12 kB of data written per ray.

Just as for memory loads, coalescing is important for memory stores. While not hitherto discussed, the intersect-and-process procedure simply loops over all primitives in a leaf; each thread tests its ray against the primitive, and writes any necessary output to a per-ray array. This may be framed as an outer loop over particles, and an inner loop over rays, the latter being executed in parallel. The implementation is not modified for packet traversal, though of course all threads in a warp test the same primitive concurrently. Unfortunately, this results in essentially no coalescing of writes; at best, writes might consist of 12 contiguous bytes if the above integral, particle index and distance are packed into a single hit result `struct`. In practice, separate arrays are more useful for later processing.

In the context of packet tracing, and with some additional bookkeeping, coalesced writes are possible. In essence, a warp concurrently tests multiple primitives against a single ray. That is, the outer loop over particles is executed in parallel, and the inner loop over rays is serial. (Using per-particle output arrays is a valid alternative, but in practice less useful.) After all ray-primitive intersections are known for the current iteration, but before any writes occur, a warp-wide exclusive scan sum (see Listing C.1, page 251) is performed over all threads which have data to write for the current ray; this produces unique, consecutive offsets for each such thread. Ensuring coalesced writes within a warp is then trivial. All threads in a warp must take care to track the total

Listing 4.9. C-like pseudo-code for a leaf traversal loop which parallelizes over primitives. `lane` refers to a thread's index within its warp.

```
for (int pi = lane; pi < leaf.size; pi += warp_size)
{
  Primitive p = primitives[leaf.first + pi];

  for (int ri = 0; ri < warp_size; ++ri)
  {
    Ray ray = warp_rays[ri];
    bool hit = intersect(ray, p);
    int offset = warp_scan_sum(hit);

    if (hit) {
      ray.results[ray.out_idx + offset] = process(p);
    }

    int total_hits = warp_count(hit);
    if (hit && offset == 0) {
      warp_rays[ri].out_idx += total_hits;
    }
  }
}
```

number of writes, even when they are not taking part. The procedure is demonstrated in Listing 4.9.

On Fermi architecture devices, the necessary ray sharing is implemented as a shared memory store for rays, allowing a threads to load rays 'belonging' to other threads. On Kepler and more recent architectures, instead the CUDA built-in `__shfl()` instructions are used. These allow in-register values to be communicated directly between threads in the same warp.

The primary disadvantage of this method is that it will lead to increased warp divergence. Parallelizing the loop over particles results in idle threads unless the leaf size is a multiple of the warp size. At $\phi_{\max} = 32$, the actual distribution of leaf sizes peaks at approximately 25, and in a majority of cases a warp is operating at an efficiency of $\lesssim 80\%$ during leaf traversal. Further bookkeeping allows for improvement in this area, but was not investigated. (Specifically, note that in total there are `leaf_size` $\times$ `warp_size` intersection tests to perform; this may be framed as a single loop, where `warp_size` consecutive elements are worked on in parallel.)

Table 4.11. Wall-clock trace time for both coalesced and non-coalesced writes. The right-most column gives the coalesced time divided by the non-coalesced; smaller values correspond to a greater improvement. For each ray-particle intersection, particle index (`int`), integral (`float`) and distance to intersection (`float`) are stored. 38 400 rays are traced through a 128^3 particle SPH, $z \sim 20$ dataset, producing 5.77×10^7 intersections.

Device	Non-coalesced (ms)	Coalesced (ms)	Fractional runtime
M2090	366	161	0.44
GTX 670	212	127	0.60
K20	227	123	0.54
GTX 960	163	110	0.67

Performance gains of approximately a factor of two observed across all tested devices, as shown in Table 4.11.

While not strictly an improvement in coalescing, using CUDA surface — a writeable texture — was also investigated. No significant increase in performance was noted.

Ray-AABB intersection tests

It is clearly important that the ray-AABB intersection test be efficiently implemented. In the context of CPU ray tracing, there exists a simple, fast and robust method due to Smits (1998), further elaborated on by Williams et al. (2005), and henceforth referred to as the *Williams* method. More complex algorithms have been put forward by Mahovsky and Wyvill (2004), henceforth the *Plücker* method, and notably used in SPHRAY (Altay et al., 2008), and the *ray-slopes* method of Eisemann et al. (2007), used by Forgan and Rice (2010). While all methods, as presented, rely on branching, the latter two involve large switch-case statements based on ray direction classification (a function of the signs of each of the direction components of a ray). Such branching will lead to significant warp divergence when consecutive rays do not have identical classifications.

The Williams method can be recast such that no explicit branching is required. Instead, `max` and `min` operations are used. This was in fact presented in Chapter 3, Section 2.1 (page 40) and is henceforth referred to as *Williams-branchless*. Aila et al. (2012) presented a further improvement, henceforth the *Aila* method, suggesting that some operations be replaced with the rather esoteric video instructions available on CUDA devices. For two functions `f` and `g`, each a `min` or `max` of two variables, the aforementioned video instructions compute `f(g(a, b), c)` in a single instruction. (Four

such instructions exist, one for each combination of `min` and `max`). A significant caveat is that they only operate on 32-bit integer inputs; however, the IEEE specification for floating-point numbers is such that, provided the input values are positive, interpreting `float` values as `int` values will result in a correct comparison (see the end of Chapter 3, Section 4.2, page 61). As shown by Aila et al. (2012), all results where input values are negative are discarded (they correspond to intersections before the ray's origin, i.e. they never result in a hit) and so their correctness is not a requirement.

Aila et al. (2012) make a further optimization when computing potential intersection points along a ray (see Eqs (3.3), page 42); along the k-axis bound of an AABB, the two such points are

$$t_{k_0} = (k_0 - O_k) \,/\, d_k, \tag{4.24a}$$

$$t_{k_1} = (k_1 - O_k) \,/\, d_k, \tag{4.24b}$$

where k_0 and k_1 are the lower and upper bounds, on the k-axis, of the AABB, and O_k and d_k are the k-axis components of the ray's origin and direction vectors, respectively. Equations (4.24) can be recast as

$$t_{k_0} = \frac{1}{d_k}\, k_0 - O'_k, \tag{4.25a}$$

$$t_{k_1} = \frac{1}{d_k}\, k_1 - O'_k, \tag{4.25b}$$

where $O'_k \equiv O_k/d_k$. The inverse of the direction components and the O'_k are constant for a ray, and may be precomputed. The above values can then be computed in a single fused multiply-add (FMA) instruction.

However, while Eqs (4.24) and Eqs (4.25) are mathematically equivalent, they do not in general produce equivalent results for finite-precision computations. Of note, results may differ in the case that $d_k = 0$. The Williams (and Williams-branchless) methods are carefully constructed to take advantage of the IEEE specification and produce correct results even when direction components are zero (Williams et al., 2005). Unfortunately, use of Eqs (4.25) removes this property. A workaround suitable for real-time graphics workloads might be to assign some small-but-nonzero value to d_k when its true value is zero; the likely-infrequent errors are an acceptable trade-off for the resulting performance gain. For the target use-cases of GRACE, the same argument

Table 4.12. The wall-clock execution time for various ray-AABB intersection test implementations. 100 000 isotropic random rays are generated, and 5000 AABBs. Rays are sorted based on the Morton key of their direction vectors. Bold values indicate the best-performing implementation for that device.

Implementation	Execution Time (ms)				
	M2090	GTX 670	K20	GTX 960	2 × Xeon E5-2620*
Williams	160	210	270	160	1500
Williams-branchless	91	140	160	**140**	2200
Aila	**90**	**130**	**150**	160	—
Plücker	130	150	180	190	**910**
Ray slopes	200	300	270	480	930

*OpenMP was used to parallelized ray-AABB tests across all 12 physical cores of two Intel Xeon E5-2620 2.0 GHz processors, with hyperthreading disabled; optimization level `O3` was set

cannot be made, and so all applicable methods implement the safe, non-FMA version Eqs (4.24).

To measure performance, all of the above methods are implemented for the CPU and GPU where possible. The core intersection functions are written once and executed on both platforms. Each method allows for some pre-computation, all of which is performed once per ray and included in the timing data. For the Williams and Aila methods, the inverse of each ray-direction component is pre-computed (three `float` values); both the Plücker and the ray-slopes methods require a ray classification based on the sign of each direction component (one `int` value); ray-slopes additionally requires gradient and intercept data (twelve `float` values).

Input rays and AABBs are identical for both CPU and GPU. 100 000 random, isotropic rays are generated, all emanating from the origin and of length 2.0. Rays are sorted according to the Morton key of their direction vectors. 5000 AABBs are also generated, specified by six co-ordinates for the upper- and lower- corners, with each such co-ordinate selected from a uniform random distribution in $[-1, 1)$. Results are presented in Table 4.12.

The CPU and GPU show almost perfectly-opposed performance characteristics. The CPU favours the more-complex, but efficient, algorithms; the GPU favours the less efficient, but non-branching variants. Somewhat surprisingly, and despite using an implementation essentially identical to that made available by the authors, ray-slopes is found to perform marginally slower than the Plücker method. This trend holds for

both increased and decreased numbers of AABBs.

The ray-slopes method performs particularly poorly on the GPU, despite being similar to the Plücker method. The primary difference between the two is the number of pre-computed values required, described above. This may be affecting compiler optimizations, particularly loop unrolling, via increased register pressure. The Aila method is notably not the highest-performing routine on the GTX 960. Consultation of the CUDA Programming Guide[21] shows that the SIMD video instructions moved from a single, native instruction to multiple instructions with devices of compute capability 5.0 and onward; the GTX 960 is a 5.2 device. In GRACE, the exact ray-AABB intersection implementation used is selected at compile-time based on the current target architecture, always selecting the highest-performing variant as per Table 4.12.

Miscellaneous traversal optimizations

Several other optimizations, which do not fit into the categories already discussed, were investigated, with varying degrees of success.

Perhaps not surprisingly, it was found that *inlining*, where a call to a function is directly replaced with the body of that function, is extremely beneficial when applied to the ray-AABB and ray-primitive intersection routines. Wherever these are used, if their definition is not available to the compiler at compile-time (i.e. the function will be linked at link-time), the traversal time can be increased by up to a factor of two. Similar guidance applies to the function which generates an AABB from a primitive, required during BVH construction.

A BVH with spheres as the bounding volume, rather than AABBs was implemented, with the motivation that a bounding sphere might more tightly contain SPH particles than an axis-aligned box. Traversal performance was reduced by a factor of two; whether or not the motivating assumption holds was not investigated.

As noted in Chapter 3, Section 4.2, the Morton key used to order primitives will result in large Euclidean distances between some positions with consecutive key values. That is, primitives with close key values may not be closely located in real space; hence, some leaves of the BVH may be quite large. Several options for improving the quality of the tree were explored. All modifications were evaluated for both cumulative

[21] Table 2, Section 5.4.1 of the CUDA C Programming Guide v8.0

and all-intersections traversal using 128^3 SPH datasets over a range of redshifts (see Section 9.2).

Perhaps most simply, the Hilbert key (see Figure 3.5, page 60) does not exhibit the occasionally-poor spatial locality of the Morton key, and in the context of LBVH may readily be used in its place. The (serial) Hilbert-key function from GADGET-2 was used to provide keys for the ALBVH builder. This did not result in any measurable increase in performance, suggesting that Morton keys are 'good enough' (for the datasets tested).

In fact, it was supposed in Section 5 that the quality of the tree is not of great importance for SPH datasets due to their relative uniformity. This supposition is easily tested with the use of a high-quality BVH. A SAH-optimized BVH implementation, which was developed by an Edinburgh student, Martin Rüfenacht, as a component of his master's thesis, again did not result in any significant increase in traversal performance. (The reduction in runtime ranges from essentially none to a few percent.) This would indeed appear to confirm my earlier assumption. Further evidence to that end is given in Section 9.4, using the OptiX (Parker et al., 2010) ray tracing package.

Nonetheless, the squared Euclidean-distance metric is used in GRACE when building the LBVH. It typically reduces runtime by only a few percent relative to pure LBVH, but is trivial to implement, while constituting only $\sim 3\%$ of the total construction time. An AABB-surface-area metric was also tested, but it tended to marginally increase traversal time, while also being more expensive to compute

In an attempt to further alleviate the GPU-hostile random-access memory pattern encountered when loading BVH nodes during traversal, texture memory has been utilized. This loads data from global memory through the texture cache, which *may* achieve higher bandwidth than standard global loads for incoherent access patterns. On Fermi-architecture devices, in the context of computing the cumulative integral along a ray, accessing nodes via a texture fetch consistently reduces the runtime by $\sim 10\%$ for both $N = 128^3$ and $N = 256^3$ SPH datasets over redshifts $z \in [20, 3]$. On newer devices, a reduction in runtime is often not observed, or otherwise is similar; runtime is never increased. This is not unexpected: with Kepler and most later architectures, where the compiler detects that data is read-only for the lifetime of a kernel it is free to replace global memory loads with loads through the texture cache. Further, the Maxwell architecture unified the L1 cache and the texture cache for caching reads from global memory.

Finally, when intersecting primitives within a leaf node, all primitives are first loaded into shared memory. This allows all particle loads to be optimally coalesced. Again in the context of computing the cumulative integral along each ray, runtime is reduced by approximately 22% on an M2090 device, 14% on a GTX 670, 21% on a K20 and 8% on a GTX 960. When instead storing information for every ray-primitive intersection, these gains are reduced to 5%, 1%, 0% and 2%, respectively. This latter use-case sees reduced improvement as it already allows for coalesced loads, provided the current leaf is large enough, as per Listing 4.9. It is important to note that the shared memory requirements increase linearly in the maximum-allowed leaf size, $\phi_{\max}$, and the size of the data type for the primitive. For SPH particles, with 256 threads per block and $\phi_{\max} = 32$, only 4 kB (of an available 48 kB) are required per block. However, larger primitive types and larger leaf sizes could easily result in a shared memory requirement that significantly reduced the occupancy, and hence performance. Conversely, for small values of $\phi_{\max}$, the overhead is likely to outweigh any gains. Primitives larger than 16 bytes also require multiple load instructions, and in general cannot be fully coalesced.

8 Verification

While naïve BVH traversal ray tracing is relatively straightforward, the fully-optimized code is not so trivial. Some attempt to verify its correctness is therefore sought.

It is straightforward to perform consistency checks on the BVH itself: every primitive should be referenced exactly once; all inner nodes leaf nodes should be referenced by exactly one parent node (excepting the root node); and the AABB of the root node should contain all primitives.

To assess the accuracy of the ray-sphere intersection routine, it is compared to a slow-but-accurate implementation; to reduce the likelihood of similar issues being present in both implementations, this reference test is somewhat different. For a ray of length l, it tests for the existence of real roots $t \in [0, l]$ of the equation (c.f. Eq. (3.7), page 43)

$$|\mathbf{d}|^2 t^2 + 2(\mathbf{O} - \mathbf{C}) \cdot \mathbf{d}\, t + |\mathbf{O} - \mathbf{C}|^2 - R^2 = 0, \tag{4.26}$$

where $\mathbf{O}$ is the origin of a ray, $\mathbf{d}$ its direction, $\mathbf{C}$ is the centre of the sphere and R its radius. This test is achieved by evaluating Eq. (4.26) on the interval $[0, l]$ via interval arithmetic; a real root exists (and, therefore, an intersection) if the resulting interval

contains zero. Note that in their typical form, quadratics suffer from the dependency problem when evaluated on an interval; this is readily solved by solving the square, i.e. recasting such that t appears only once. Hence

$$at^2 + bt + c = 0 \tag{4.27}$$

becomes

$$a(t - h)^2 + k = 0, \tag{4.28}$$

where a, b and c are defined by comparison with Eq. (4.26), $h \equiv \frac{-b}{2a}$ and $k \equiv c - \frac{b^2}{4a}$.

To address the need for increased accuracy, all computations are performed using the GNU Multiple Precision Arithmetic Library.[22] Specifically, GNU MP's `mpq_class` is used, a container for rational numbers; internally, a value is represented as numerator and denominator in its canonical (no common divisors) form. Mathematical operations are then exact, up to the precision of the initial input (which is necessarily a standard floating-point value), and themselves return canonical-form rationals.

For testing, 2×10^6 particles are randomly generated with centre co-ordinates in $[-0.5 \times 10^4, 0.5 \times 10^4]$, and radii in [80, 280]; these values very approximately simulate the properties of a 128^3 particle SPH dataset in a 10 Mpc box with 64 near-neighbours. An isotropic distribution of 32 000 rays is generated, with origin (0, 0, 0) and of sufficient length to exit the volume in which spheres may be placed. Ray-particle intersections are computed pairwise, on the CPU only,[23] to avoid any errors introduced by the acceleration structure. Of the $\sim 4.4 \times 10^7$ total intersections, only 16 GRACE results differ from the reference; this excludes impact parameters b for which $|1 - b^2_{\text{ref}}/R^2| \leq 10^{-8}$, where R is the radius of the particle, which is chosen as a limit of negligible contribution for an intersection. Both false-hits and false-misses contribute to these errors. However, in all 16 cases, we find that $|1 - b^2/b^2_{\text{ref}}| \lesssim 10^{-6}$.

With the core intersection test shown to be sufficiently correct, the acceleration structure is tested. This is achieved similarly to the above, with particles and rays generated identically. Reference results are again found via pairwise comparison on the

[22] `https://gmplib.org/`

[23] Care must be taken during compilation to ensure that the GRACE intersection routine is equivalent to that executed by the GPU; in particular, expressions containing `float` and `double` values must obey IEEE binary32 and binary64 specifications, and not be executed by the CPU's x87 FPU (floating-point unit), which is typically the default, and has increased precision relative to `double` types. This was achieved by providing the `-mfpmath=sse -msse2` flags to gcc.

Table 4.13. The ratio of GRACE estimated volume integrals, $\hat{V}$, to analytic integrals, V, for several SPH datsets. In all cases, 262 144 rays were generated.

$N_{\text{partilcles}}$	z	$\hat{V}/V$
128^3	20	1.00042
	8.5	1.00033
256^3	20	1.00040
	8.9	1.00057

CPU, but using the GRACE routine, rather than the slower interval-based test. These are compared to results obtained on the GPU, here making full use of the BVH. Results are found to match exactly. While it is entirely possible that some differences will occur under more exhaustive testing, due to spurious misses from the ray-AABB intersection routine, it seems reasonable to assume such errors would have a negligible impact in a typical use-case.

Finally, all relevant aspects of the code are tested via computation of the volume integral of the SPH kernel function — see Eq. (2.31) — for several GADGET-2 datasets. The volume integral of the SPH kernel is equal to one, and hence the volume integral over a dataset, V, is equal to the number of particles. This volume integral is estimated, using GRACE, as $\hat{V}$: the sum of 262 144 line integrals multiplied by a per-line effective cross-sectional area. Rays originate from a common plane, lie along the z-axis, and begin and end outside the simulation volume. The common origin-plane is divided into evenly-spaced cells, with each ray's origin corresponding to the centre of a unique cell. Finally, the x and y bounds of the plane are equal to those of the box bounding all particle volumes, rather than particle centres. The value of $\hat{V}/V$ for several test datasets is given in Table 4.13, and in all cases shows an error of $\lesssim 0.1\%$.

9 Performance

Some performance data has already been presented during the discussion of the various optimizations employed and developed for GRACE. In this section, more extensive performance testing is presented. Small and medium-sized SPH datasets, covering a range of redshifts relevant to cosmological radiative transfer, are used. To put the results into context, GRACE is compared to two CPU implementations developed by

other authors, and to OptiX, an application framework for ray tracing on CUDA GPUs from NVIDIA (Parker et al., 2010).

Performance measurements for GRACE use the `cudaEvent_t` type and related functions provided with the CUDA toolkit; this allows for synchronization with kernels, which otherwise execute asynchronously. Start and end points for execution timing are typically close to the launch of the underling kernel, and include negligible overhead. Memory allocations and transfers to or from the GPU are generally not included. When profiling CPU codes, similar practices are followed, and memory operations not central to the task being profiled are not included. All OptiX performance measurements give the time taken for an `optix::ContextObj::launch` call to return. An additional warp-up run, which is not included in the timing data, is performed for OptiX. This hides any first-call overhead, which can be significant, as OptiX is implemented as a just-in-time compiler. It also triggers a build of the acceleration structure.

All compilations set the optimization level flag `-O3` under GCC version 4.9.4; OptiX version 4.1.0 is used;[24] and GRACE and OptiX additionally use the CUDA toolkit version 8.0. Note that OptiX 4.0 no longer support Fermi-architecture devices, thus the Tesla M2090 used here is not supported. All quoted results are the mean of 10 iterations.

9.1 CPU Implementations

Both CPU implementations were developed specifically for ray tracing through SPH datasets. The first is a stripped-down version of the SPHRAY radiative transfer code (Altay et al., 2008), such that only ray-particle intersections are found.

The second is an SPH ray tracing code developed by Martin Rüfenacht, RTSPH, as part of his masters project, under the supervision of Eric Tittley. This code builds a SAH-optimized BVH, in serial, while ray traversal is parallelized using OpenMP.

Further implementation details are given in the following sections. Note that while both codes use an acceleration structure to accelerate the traversal, they have not received significant further optimization efforts, such as that carried out for GRACE in Section 7.3. As such, they are not an entirely fair comparison. Nonetheless, they represent two important performance points researchers may expect to attain: (i) from

[24] Current as of 2017-05-01.

Table 4.14. The SPH particle counts and redshifts of the datasets used for assessing ray tracing performance, along with their identifiers.

Identifier	$N_{\text{particles}}$	z
n128-z3	128^3	3.000
n128-z8		8.459
n128-z11		10.96
n128-z20		19.68
n256-z3	256^3	3.000
n256-z9		8.851
n256-z12		11.74
n256-z20		20.33

a currently available code (SPHRAY); and (ii) from developing their own code, following recent literature in the field of computer graphics (RTSPH).

9.2 Test Datasets

Several test SPH datasets, of varying particle counts and redshifts, have been used to assess performance. Each dataset is a snapshot from one of two runs of the cosmological SPH simulation code GADGET-2 (Springel, 2005). Both runs begin with pristine initial conditions, at $z \sim 166$, in a box with side-length of 10 Mpc (comoving); both set a target near-neighbour particle count of 64; and both proceed with periodic boundary conditions and comoving integration to a redshift of $z \sim 3$. WMAP7 Komatsu et al. (2011) cosmological parameters were set.

The smaller dataset consists of 2×128^3 total particles, equally split between dark matter and SPH; the larger consists of 2×256^3, again with an equal dark matter-SPH split. (Only the SPH particles are ray-traced.) The particle counts and redshifts of the samples used here are codified in Table 4.14.

9.3 Tree Build Performance

The time taken for each code to build its acceleration structure is given in Table 4.15. For GRACE, this is an ALBVH with a Euclidean-distance metric; for OptiX, a TRBVH (Karras and Aila, 2013); for RTSPH a SAH-optimized BVH based on Wald (2007); and for SPHRAY, an octree.

As already noted, both the RTSPH and SPHRAY builders execute in serial on the CPU.

They are therefore expected to perform quite poorly relative to the optimized GPU implementations of GRACE and OptiX. Note also that SPHRAY typically makes several attempts at a tree-build before succeeding, as it initially underestimates the memory requirements of the octree; here, the reported runtime includes only the successful iteration.

The TRBVH structure of OptiX was presented in Karras and Aila (2013). It incorporates the LBVH builder of Karras (2012) and applies a post-processing stage: small sub-trees are re-organized in an effort to improve the structure's ray tracing performance. It is the fastest builder available in OptiX (for the version used here), but is stated to result in similar ray tracing performance to the highest-quality CPU BVH available (which is based on Stich et al. (2009)).

Building the acceleration structure in OptiX is implicit, executed by the framework as needed. To measure construction time, a re-build of the BVH is forced with a call to `optix::AccelerationObj::markDirty` before each traversal; the traversal is then timed. A second run, which is otherwise identical to the above but which does not force a rebuild of the structure, is then performed and also timed. The TRBVH build time is taken to be the difference in these two measurements.

The OptiX measurements given in Table 4.15 almost certainly include memory allocations and copies; for this reason, the bracketed GRACE results also give the total run time. This 'total run time' includes all necessary memory copies to the GPU, including transfer of the SPH particles themselves. It also includes the time taken to compute the x, y and z bounds of the simulation volume, which are needed when calculating the Morton keys (see Chapter 3, Section 4.2); this is not included in the execution-only GRACE results as bounds are typically provided with the input dataset.

As expected, GRACE performs exceptionally well relative to the CPU codes. Even if the builders for both RTSPH and SPHRAY were optimally parallelized and run across all six cores of the Xeon E5-2620 used for testing, their runtimes would remain approximately an order of magnitude greater than those of GRACE, but would be comparable to OptiX.

Similarly, even if we assume some significant overhead is included in the OptiX results, GRACE remains competitive. This holds true for a maximum leaf size of $\phi_{\max} = 1$, a poor-performance point for GRACE: on a GTX 670, for example, $N = 128^3$ dataset build times increase to ~ 85 ms for execution-only, and ~ 95 ms for total runtime. As an interesting comparison, OptiX also makes available a "refit" rebuild, which modifies

Table 4.15. Time taken by each code to build its acceleration structure. GRACE builds an ALBVH, OptiX a TRBVH, RTSPH a SAH-optimized BVH and SPHRAY an octree. Dashes denote a build failure (due to insufficient memory in all cases). GRACE quotes both execution-only and, in brackets, total time.

Identifier	Build time / ms							
	GRACE				OptiX		RTSPH	SPHRAY
	M2090	GTX 670	K20	GTX 960	GTX 670	K20	Xeon E5-2620	
n128-z3	21.1 (38.3)	16.4 (24.1)	18.1 (33.9)	19.1 (28.7)	194	203	2110	1240
n128-z8	21.1 (39.3)	16.3 (24.0)	17.7 (33.1)	18.6 (26.9)	184	195	2030	824
n128-z11	21.1 (39.2)	16.5 (24.4)	17.7 (33.4)	18.6 (26.9)	180	191	2020	816
n128-z20	21.0 (39.0)	17.7 (26.0)	16.1 (30.7)	18.7 (27.0)	178	188	1950	804
n256-z3	156 (281)	—	115 (218)	—	—	1570	18 700	11 000
n256-z9	154 (281)	—	114 (217)	—	—	1500	23 000	8320
n256-z12	153 (280)	—	113 (217)	—	—	1480	23 000	7120
n256-z20	153 (278)	—	113 (218)	—	—	1460	18 900	7070

only the node AABBs, leaving the hierarchy unchanged. While no details are given, this is likely implemented as a tree-climb similar to that used for ALBVH construction in GRACE. The execution time for this refit procedure, again on a GTX 670 and for $N = 128^3$ datasets, is $\sim 15\,\text{ms}$.

Thus, while comparison to published literature values in Section 5.3 (Table 4.3, page 89) suggested runtimes might yet be decreased by an order of magnitude, such performance may be practically unattainable for real-world codes.

9.4 Ray Tracing Performance

The functionality of the CPU codes differs, and so two separate comparisons are made. RTSPH outputs cumulative per-ray column densities; the OptiX implementation is similar, borrowing the equivalent routines from GRACE (user-supplied functions are identical wherever possible). Timing results for these two codes and the equivalent GRACE function are given in Figure 4.19.

Rays are generated similarly as for volume-integral estimation (see Section 8). All rays are parallel, travel in the $+z$ direction, originate and terminate outside the simulation volume, and cover a large enough region that all particles may be intersected. OptiX results include ray-generation time, a restriction of the framework; however, for GRACE, this step is negligible and its exclusion does not materially affect the results.

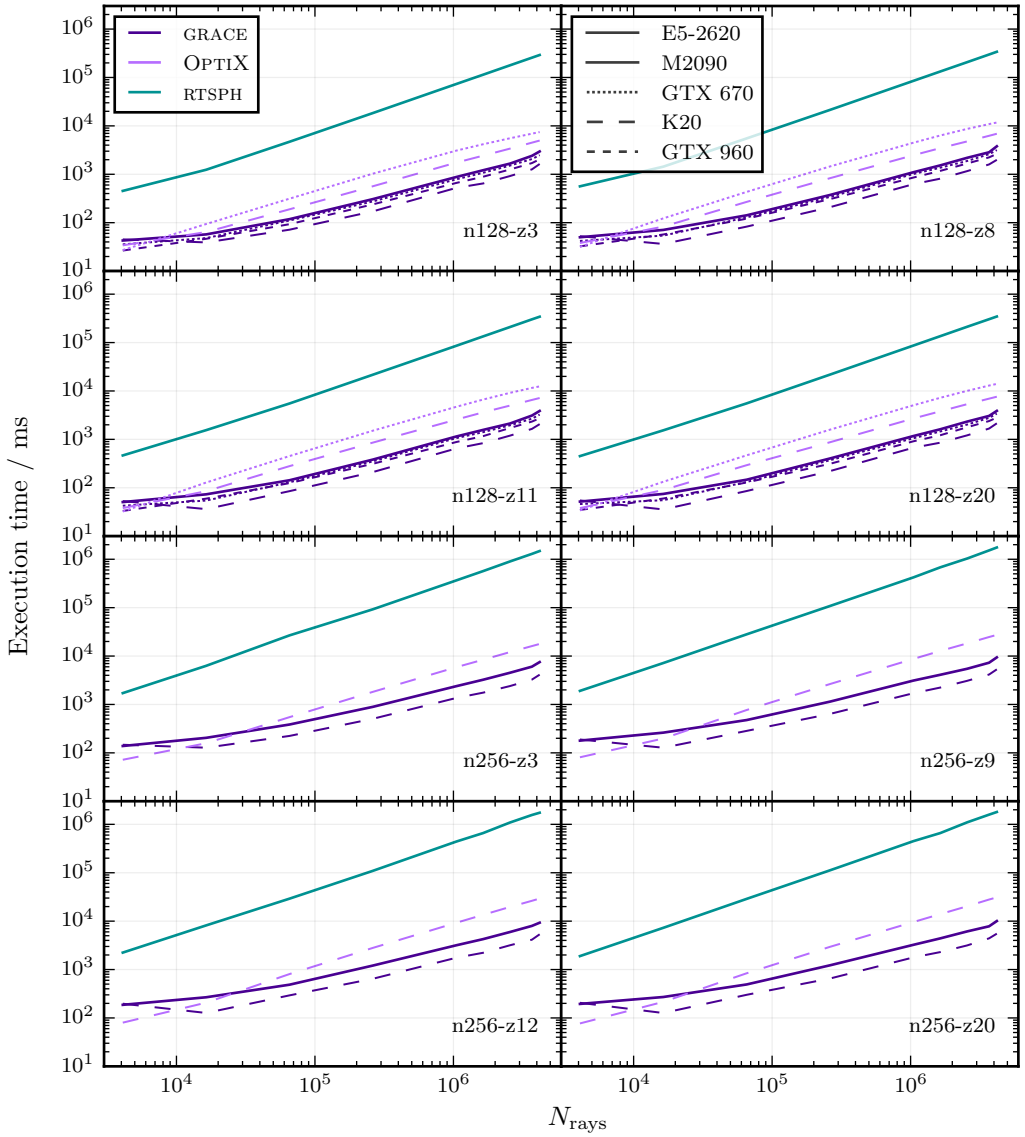


Figure 4.19. Execution time to accumulate the SPH kernel integral along each ray. All rays are parallel, travel in the z-direction, and originate and terminate outside the simulation volume.

RTSPH runs were parallelized with OpenMP, executing across all twelve cores of two six-core Intel Xeon E5-2620 CPUs, with hyperthreading disabled. Note that all tested GTX-class GPUs failed for all $N = 256^3$ datasets due to insufficient memory. Further, the Tesla M2090 is no longer supported by OptiX, and while the GTX 960 is supported, it was not tested.

As when measuring BVH construction times, the TRBVH acceleration structure of OptiX is used. Comparison of ray tracing runtimes for TRBVH and the higher-quality SBVH show reductions for the latter on the order of a few percent. Further, comparison of runtimes for SBVH and the lowest quality BVH, which is somewhat akin to LBVH, similarly show only percent-level improvements for SBVH. In contrast, high-quality BVHs often approximately halve the runtime when traversing computer graphics scenes (see e.g. Aila et al., 2013, Tables 1 and 3). This is further evidence that BVH quality is not particularly important for cosmological SPH datasets.

Both GRACE and OptiX outperform RTSPH, as expected. It is notable that this holds down to $\sim\lesssim 10^4$ rays, where the scaling performance of GRACE begins to plateau, even increasing for the K20. At larger problem sizes, and well into the effective-scaling regime for GRACE, the performance difference of GRACE and RTSPH is approximately two orders of magnitude.

Comparing to OptiX, we see more modest gains in the approximately-linear scaling region; the difference is roughly a factor of four at 1×10^6 rays. We also see sub-linear scaling for both GRACE and OptiX at high ray counts, where we naturally achieve greater coherence between rays within a warp. This effect is strong for GRACE due to its use of packet traversal. On the other hand, for $N_{\text{rays}} \lesssim 10^4$, OptiX achieves shorter runtimes, with near-linear scaling down to at least 4×10^3 rays. These properties are further elaborated on in Section 9.5.

Turning back to GRACE, note that a plateauing of the scaling relationship is actually expected for low ray counts. The GTX 670 device, for example, has 7 multiprocessors (SMs) in total, and the tested implementation caps the number of warps resident on a single SM at 40. The device therefore supports 280 concurrent warps, or 8960 threads, corresponding to 280 packets and 8960 rays, respectively. Some significant fraction of these available threads must be active to hide the latency inherent to the implementation, so perfect scaling for $N_{\text{rays}} \lesssim 8 \times 10^3$ is not expected. (This threshold varies by device, but is $O(10^4)$ for all those tested, and in fact all currently-available GPUs.) On a GTX

670 device, and for the n128-z8 dataset of Figure 4.19, the NVIDIA profiler shows that the achieved occupancy is a mere 23% at 4×10^3 rays (with a theoretical maximum of 62.5%), increasing to 50% for 8×10^3 rays.

Of course, this does not explain the increased runtimes seen for the K20, nor necessarily the severity of the plateauing. Again, this is expanded upon in Section 9.5.

Moving on to a comparison with SPHRAY, note that it stores per-particle column densities (i.e. per-particle integrals) and distance-along-ray for each ray-particle intersection, and sorts them from closest-to-farthest along the ray. No further processing is carried out by the version used here, and the code executes in serial at all stages. The GRACE implementation computes and stores per-particle column densities, distance-along-ray and particle index, and again these intersection data are sorted from closest-to-farthest along the ray. Both codes generate rays originating from the centre of the simulation volume, with a random direction vector, and of sufficient length to exit the volume. Timing results are shown in Figure 4.20.

Due to the volume of data produced, the GRACE implementation is limited to $\sim 10^5$ rays per kernel launch for 128^3 particle datasets, and lower for 256^3 datasets. Again, no GTX-class device is able to build an acceleration structure for the 256^3 datasets and therefore cannot be profiled.

Results here are largely similar to the previous comparison to RTSPH. For low ray counts, GRACE is approximately a factor of two faster than SPHRAY; for larger ray counts, the difference is approximately two orders of magnitude. Again, we see that GRACE suffers from poor, and even negative, scaling as the problem size is reduced. The reasons are as discussed above, and again further detailed in Section 9.5. The effect is exaggerated relative to Figure 4.19 due in part to the reduced x-axis scale, but primarily because this form of traversal is more strongly affected. This use-case in particular makes clear the need for a future optimization effort to improve scaling at smaller problem sizes.

9.5 The effectiveness of packet traversal and limits to performance

Figures 4.19 and 4.20 highlight a potentially surprising feature of the packet-based traversal in GRACE: the total runtime can increase as the number of rays is reduced. Further investigation shows that the effectiveness of packets varies greatly with the

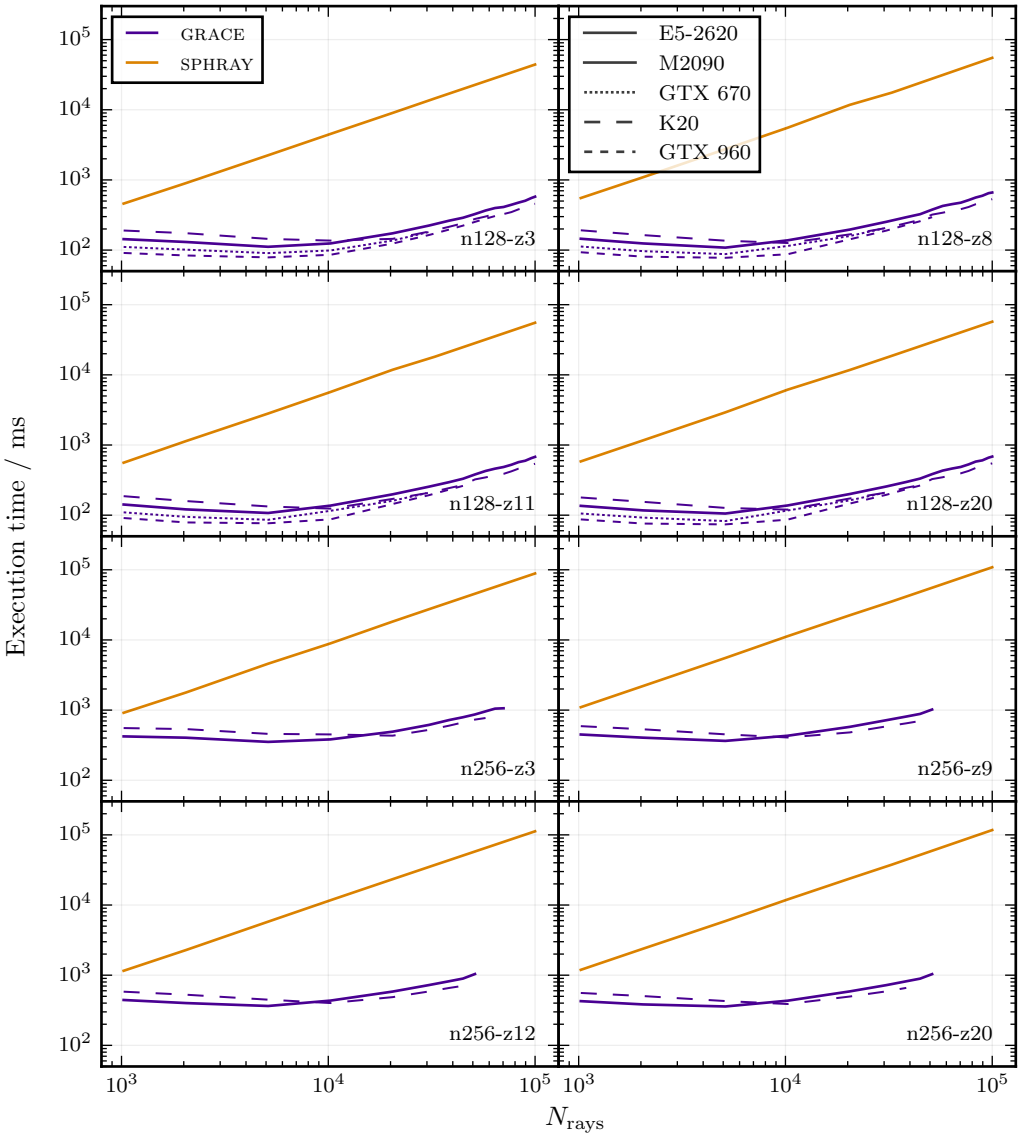


Figure 4.20. Execution time to compute the SPH kernel integral for each intersection along each ray, and sort all intersections by distance along the ray. All rays originate in the centre of the simulation volume with a direction vector selected from a uniform-random distribution, and all rays exit the volume.

number of rays and the dataset. This is demonstrated below, and some attempt is made to explain the behaviour.

That fewer rays might result in longer execution times is not necessarily surprising when one considers that decreasing the total number of rays *with a common origin* also increases the size of the cone bounding each packet. (Recall that rays are sorted such that those with similar directions are close in memory, and thus likely to be in the same packet.) With sufficiently large bounding cones, packets will force all rays within them to traverse substantially more nodes of BVH than they otherwise would, thus increasing the execution time. As an extreme example, imagine a packet whose rays' directions cover an entire hemisphere; clearly, one would not expect packet traversal to be more effective than independent-ray traversal in this case.

More generally, this reasoning also suggests that the effectiveness of packet traversal will vary with the input dataset. If the size of the primitives is reduced, relative to the size of the volume, so too are the AABBs of nodes, and packets must be accordingly smaller to maintain effectiveness. This behaviour was visible to some extent in Figure 4.17 (page 124): there, both larger values of N_{ngb}, and smaller values of N at fixed N_{ngb}, resulted in relatively-larger gains for packet traversal over non-packets. One will note that both of these variations are equivalent to increasing the smoothing radii relative to the size of the simulation box, and thus decreasing the size of a packet relative to the size of the primitives.

The placement of the source then also becomes a factor. If located at the box centre, all effective ray lengths are $\sim L_{\mathrm{box}}/2$; if at the corner, some rays will intersect particles out to $\sim L_{\mathrm{box}}$. This is significant, as a doubling of a packet's length results in a four-fold increase in the area covered where it terminates. In fact, for low ray counts, box-corner sources can increase packet traversal runtimes, because packets become so inefficient wherever they extend through the length of the box. Conversely, runtimes for non-packet traversal are reduced, because a large number of rays do not even enter the simulation volume and immediately exit their traversal.

These observations show Figures 4.19 and 4.20 to be somewhat misleading, containing a single and centrally-placed source. For example, if we were to trace 1024 rays from 16 (near-centre) point-sources, we would not expect the *per-ray* performance achieved for 1.6×10^4 rays, but rather that of 1024 rays. Similarly, if the source were to be moved, we may see a reduction in performance — and, crucially, one which is not

expected for the other codes.

The situation is further compounded since, as previously noted, low ray counts lead to low occupancy, and hence a plateauing of the runtime. In the above example, we might then actually expect the per-ray performance achieved to be greater than that of 1024 rays for our 16-point source, because while packet effectiveness is equally low, the device will be better-able to hide latency.

The confounding effect of low occupancy can be marginalized by tracing rays from multiple sources simultaneously. If these source locations are distributed throughout the box, we will also average over the effects of source placement. Several well-distributed source locations are desirable in any case, since it better represents the typical use-case in radiative transfer problems. To that end, in Figure 4.21 the execution time *per source* is shown, plotted as a function of the number of rays *per source* for several source counts. To put previously-reported results into context, a (potentially low-occupancy) source count of one is also given, where the source is centrally placed. For all other source counts, source locations are chosen at random from within the simulation box. Since $N_{\text{sources}} = 10$ is still relatively few, and we wish to avoid the source placement affecting the results, these measurements are actually an average over 100 sources traced in batches of 10. And, because individual-ray traversal is expected to be more effective both for lower ray counts and, at a given ray count, for source locations away from the box centre, the two non-packet implementations of Figure 4.17 are also tested. Finally, the datasets are also those of Figure 4.17, because both the number of particles and the number of near-neighbours are expected to be of significance.

First, note that per-source runtimes for the single, centrally-placed source are always slightly higher. This is expected. Once a value of N_{rays} per source is reached that allows packet traversal to be effective for all source positions, we see shorter runtimes for source locations closer to the box corners, because rays, on average, exit the volume earlier. That is to say, once packets are effective, source placement has an identical impact on both packet and independent-ray traversal.

There is a clear break in the effectiveness of packet tracing at $\sim 10^4$ rays for all datasets. That the location of this feature is constant regardless of the dataset implies that it is driven primarily by low occupancy rather than simply that packets have become inefficient. This is supported by noting that the crossing points of the independent ('No packet') and independent with postponed leaf traversal ('No packet + postpone') also

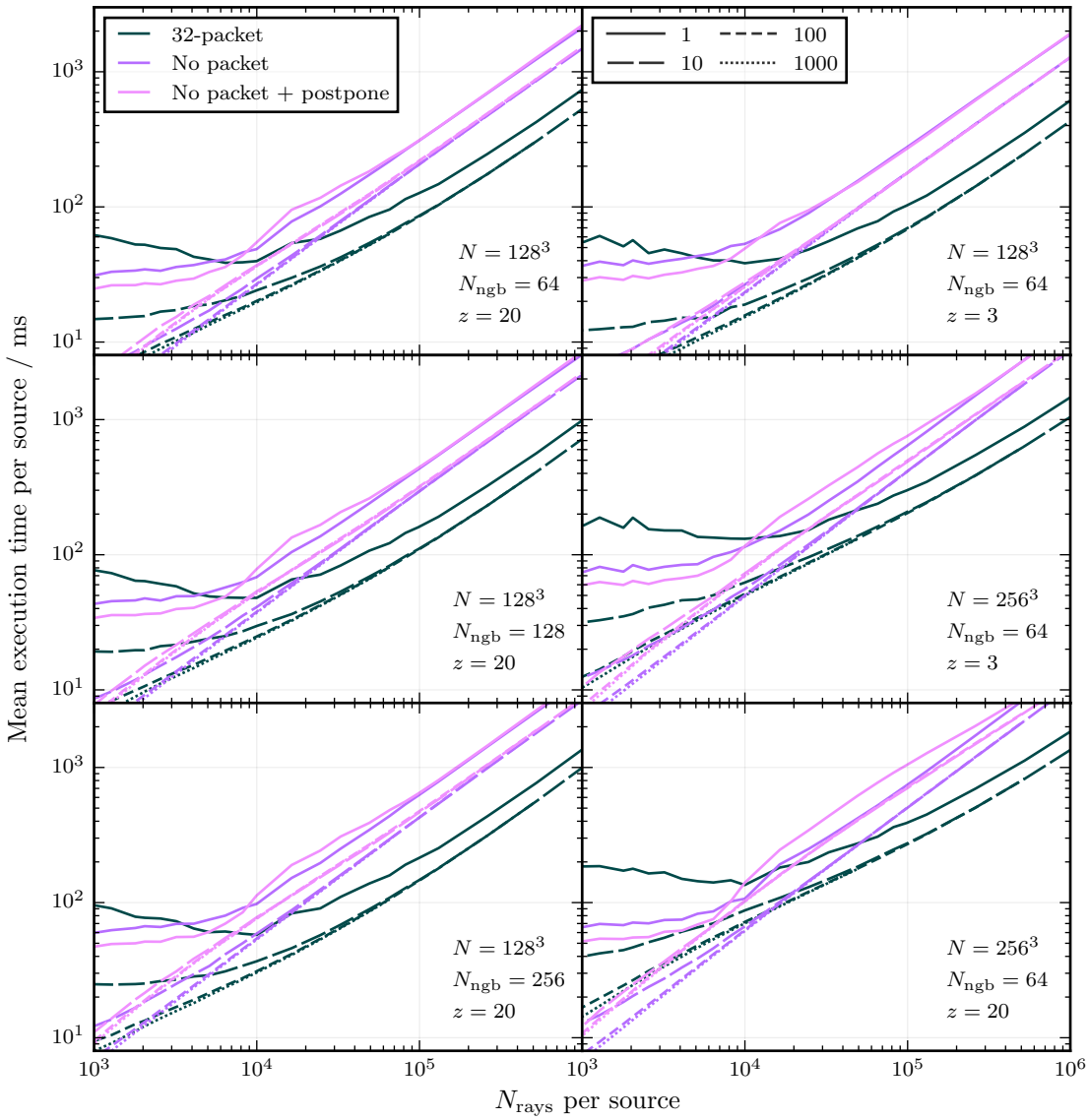


Figure 4.21. Mean traversal performance per source, for various numbers of rays per source (along x-axis) and numbers of sources (dashed lines). Packet tracing, independent-ray tracing, and independent-ray tracing with leaf traversals postponed are shown. Sources are placed at uniform-random locations within the box, except the single-source, which is centrally placed. In all cases, a maximum leaf size of $\phi_{\mathrm{max}} = 32$ was used. The total SPH-particle integral was cumulated along each ray. N refers to the number of particles, N_{ngb} the number of near-neighbour particles (which has been artificially increased from 64), and z the redshift. All results were obtained from a Tesla M2090 (Fermi) device.

occur here; postponed leaf traversal is designed to increase warp execution efficiency, and thus mitigates low occupancy.

Let us now define the point at which packet traversal becomes effective to be the value of N_{rays} per source for which it attains equal runtime to the independent-ray implementations. Again, this value decreases as N_{ngb} increases, and increases as N increases. Simultaneously tracing rays from a greater number of sources, even at constant N_{rays} per source, has a mildly positive effect for $N_{\text{sources}} \lesssim 100$. This can be attributed to the hardware only reaching peak performance for large workloads. One often finds that peak performance, when measured as number of elements (here rays) processed per unit time, tends to be achieved at problem sizes which far exceed those necessary for high occupancy. It is, for example, also seen in Thrust algorithms. The exact causes are hardware-dependent, but can typically be thought of as reducing the relative cost of constant-time overheads, such as kernel initialization and populating the caches.

For $N_{\text{sources}} \gtrsim 100$, the crossing point is approximately constant for a given dataset, and we interpret this as the 'true' limit for packet effectiveness.

Attempting to develop a well-motivated heuristic for these crossing points is, however, challenging. One might make a geometric argument, supposing that packets are only effective if their cross-sectional area is comparable to that of the AABBs of the leaf nodes across some significant fraction of the ray's length. Assuming a 32-ray packet's solid angle to be approximately $32 \cdot 4\pi/N_{\text{rays}}$, and taking the AABB cross-section to be $\langle V \rangle^{2/3}$, where $\langle V \rangle$ is the mean leaf bounding box volume, one expects packets to be effective only for $N_{\text{rays}} \gtrsim 1 \times 10^5$ for $N = 128^3$, and $N_{\text{rays}} \gtrsim 4 \times 10^5$ for $N = 256^3$ (for $N_{\text{ngb}} = 64$ in both cases). Comparing to Figure 4.21, while not terribly inaccurate, this is clearly too conservative.

It is also interesting to note that the previously-observed increase in runtime at low ray counts occurs only for packet tracing, and only for a single source. While not entirely clear, this might be explained as a further benefit of the higher source-count runs filling the device: in addition to achieving higher occupancy, caches are better populated. An isolated set of packets, as in the single-source case, always incur high latency when loading new nodes. Packets operating on a filled device, however, may find that those nodes already exist in cache, having been recently loaded by another warp. Hence their latency, and overall runtime, is reduced.

In this vein, some discussion of the effectiveness of packet traversal from the per-

spective of the hardware is warranted. It seems initially paradoxical that it should be beneficial at all: under packet traversal, a ray (nearly) always visits more nodes than it would otherwise, and can never visit fewer nodes. Yet runtimes are reduced. This implies that equal or more work has been performed in less time. Such a result can only be a consequence of reduced latency on memory loads or increased warp execution efficiency.

We begin with the latter, since it is easier to quantify. The state of every warp at each stage of traversal is recorded, henceforth a *traversal event* (TE). These TEs include the number of active threads, N_{active}, within the warp. Since all threads in a packet are always active, there 'active threads' are instead defined to be those which actually intersect the current node. These data are taken from a run with a single, centrally-placed source and the $N = 128^3$, $z = 20$ dataset, with 32 000 rays. This configuration is known to be effective for packet traversal.

It is found that packet traversal moderately improves the execution efficiency during inner node traversal, but marginally degrades it where leaf nodes are being processed. Recalling from Figure 4.17 that the intersection procedure, i.e. leaf traversal, typically contributes most to the total execution time, this does not seem to be a worthwhile trade-off. Postponed-leaf traversal is largely comparable to independent ray traversal during node traversal, but results in excellent warp execution efficiency during leaf traversal. Over 78% of TEs attain the ideal $N_{\text{active}} = 32$, compared to only 26% for independent ray traversal, and 12% for packet traversal. This would seem to be a much more desirable balance. However, returning again to Figure 4.17, we see that postponed traversal results in the longest leaf-traversal times, showing a reduction only in processing time.

We must thus conclude that increased warp execution efficiency is not the reason for the success of packet traversal, and in fact can be harmful. To reiterate, the only other mechanism by which packets might reduce execution time for the same workload is by reducing latency for data loads (and, possibly, for instruction loads). In Section 7.3 it was supposed that the benefits are due to better coalescing of memory loads. Certainly this is a factor, but it is not a complete picture.

In fact, under packet traversal we see a significant reduction in the total number of memory load requests, and hence the number of requests in flight at any one time; this will in turn reduce cache thrashing (where elements are moved in and out of cache

so rapidly that the cache is not effective). In cases where rays in a warp *generally* traverse the same nodes, packet traversal maintains this synchronization even where minor deviations would otherwise cause rays to de-synchronize in their traversal. This in turn reduces the number of times a node, or a set of primitives, can be loaded by the same warp to a maximum of one.

To demonstrate the potential gains of this behaviour, we can examine the number of times any given node is requested by a warp. (If multiple threads in a warp request the same node *at the same time*, it counts as a single request.) The TEs used previously also contain the index of the node currently being requested by each thread. For each warp, we count the total number of requests made for each node, provided it is requested at least once. During non-packet traversal, this count can be between 1 and 32 (the size of a warp) inclusive, but is always equal to one for packet traversal.

Only requests for leaf nodes are counted, since leaf traversal dominates the execution time. Further, loading of a leaf node also entails loading of all its contained primitives, amplifying the inefficiency inherent to requesting the same leaf multiple times. In any case, this does not materially affect the findings. A histogram is produced for the counts, where the bins are the possible count values, [1, 32], and the y-axis is (proportional to) the number of times this count occurred. Thus, we estimate how frequently, on average, a single node is loaded on $n \in [1, 32]$ separate occasions by a single warp. The results from 48 randomly-selected warps are shown for various input datasets in Figure 4.22. Again, 32 000 rays are traced in total, from a single, central source.

For datasets where packets are most effective (c.f. Figure 4.21), warps generally request the same leaf nodes many times; packets are able to significantly reduce redundant loads in these cases. Conversely, where packets are less effective, warps are more likely to request a leaf only once.

This effect is seen more directly through metrics provided by the NVIDIA profiler. For example, on a GTX 670 device, for the $N = 128^3$, $z = 20$ dataset, the number of uncached memory loads is reduced by a factor of ten under packet traversal. The number of reads from L2 cache also falls, but only by a factor of five, so the ratio of cached-to-uncached loads approximately doubles. The number of reads through the texture cache — through which only nodes are fetched — increases by about 10%. This is a per-thread counter, so shows that, for this configuration and dataset, the number of unnecessary node traversals constitutes only about 10% of the total. (This includes

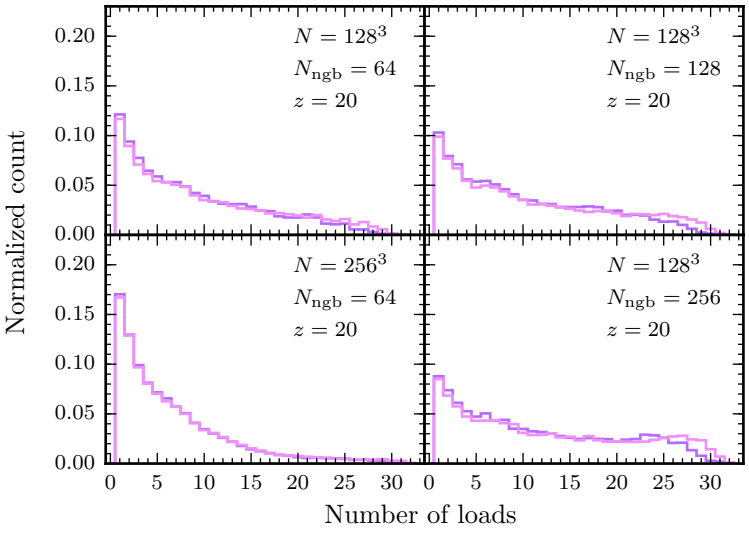


Figure 4.22. Histogram of the number of times a warp requests the same leaf under independent-ray traversal. Independent-ray traversal (lilac) and postponed-leaf traversal (pink) are shown.

both inner and leaf nodes; the fraction of unnecessary leaf traversal will be higher.)

In summary, it has been shown that packet traversal is effective because it substantially reduces the demands on the memory subsystem. This outweighs the cost of forcing rays to enter nodes of the BVH which they do not intersect, far more so than might be expected. A geometric argument which supposes packet traversal will only be effective when the cross-sectional area of a packet is always comparable to, or less than, the cross-sectional area of a leaf's AABB is far too conservative. Execution efficiency has also been shown to be of little importance. Doing less work per active thread per clock cycle can be beneficial if it reduces the number of memory requests or improves caching. Combining this analysis with the result from Figure 4.17 that leaf traversal contributes most to the total execution time for large leaf sizes, we ultimately conclude that packet traversal is effective primarily because it reduces the number of times a warp must load the same primitives.

Nonetheless, there remain situations where independent-ray traversal is optimal, as demonstrated in Figure 4.21, such as when low coherence between rays is unavoidable. Moving to non-packet traversal there achieves performance gains of up to a factor of ~ 3. Estimating which method will be most effective remains an open question, but is easily solved by performance profiling on a case-by-case basis.

Specific implementations aside, the NVIDIA profiler finds all kernels, packet tracing or otherwise, to be latency-limited. This can be mitigated by doing more work per unit

byte read from memory. Unfortunately, the BVH structure is already fairly compact, and the size of a primitive cannot in general be reduced. We can, however, attempt to ensure that a requested node is more likely to result in an intersection, and similarly that the fraction of intersected primitives within a leaf is maximized. This is equivalent to improving the quality of the BVH. Of course, an improved BVH was previously found to have little impact, but that claim now warrants further investigation. For one, intersection tests form a significant fraction of the traversal time, but for the large leaf sizes which are optimal here the fraction of primitives within a leaf a ray actually hits is small ($\lesssim 20\%$). OptiX results also support this suggestion: runtimes are decreased by up to 20% when using its high-quality TRBVH builder (for which results were presented) compared to a lower quality BVH.

Looking beyond SPH datasets, for low values of $\phi_{\max}$ it is also possible that node traversal will become the dominant component. By artificially increasing the size of a node (representing them as four `double4` rather than four `float4` elements) and measuring only the time for node traversal, it is found that runtime scales linearly with the size of the node, even for packet traversal, and thus node traversal is bandwidth limited. This has potential implications for other datasets where leaf sizes may be smaller.

Recent work by other authors has focussed on compressing the node structure further, for example by increasing the number of children per node, and expressing child bounding boxes as a fraction of their parents in reduced precision (e.g. Ylitie et al., 2017, and references therein).

10 Generalizing GRACE to other datasets

While the algorithms and implementation of GRACE has thus far been framed in the context of ray tracing SPH datasets, it is applicable for any volume element or primitive which, assuming Euclidean geometry,

i) can be checked for intersection with a ray; and
ii) allows for computation of an axis-aligned bounding box.

Ideally, both of these requirements will be implemented in a computationally efficient manner, the former in particular.

As a practical matter, I have identified several other components which I expect to

be needed for other geometries — and which in fact may be relevant for SPH under different use-cases:

i) primitive centroid computation (for Morton keys);
ii) processing of found intersections;
iii) pre-processing of rays immediately before traversal (for example, to compute values which aid ray-primitive intersection routines);
iv) per-ray data to be maintained during a ray's traversal (for example, to accumulate a value along a ray); and
v) post-processing of rays which have completed their traversal (for example, to write per-ray data to global memory).

(Note that while primitive centroids can be approximated as the centroid of the always-required primitive AABB, in some cases — such as for a sphere — the centroid may be more efficiently or more accurately computed separately.)

In GRACE, the required flexibility is exposed via C++ templates. All of the above-enumerated components, with the exception of the per-ray data, may be provided as a functor, a C++ class which defines the `operator()`, and hence may be called like a function. The arguments and return value for the various components are fixed, and sufficiently broad that most use-cases should be covered. The BVH construction and traversal functions accept these functors as template arguments. Users are therefore entirely free to provide their own implementations, while still benefiting from the acceleration structure and highly-optimized traversal routine. The per-ray data is also a template, but a simple `struct`.

GRACE provides wrapper functions to implement the SPH traversals already mentioned: cumulating integrals along each ray, and providing all ray-particle intersection data (particle index, distance to intersection and integral).

As a final detail, users may also specify, as a function argument, a shared memory size requirement, which is allocated at runtime (functionality available as part of the CUDA kernel launch syntax). A corresponding initialization functor is also accepted, and is, for example, used in GRACE to copy the SPH kernel lookup table to shared memory, once, before any rays are loaded.

While certainly flexible, this system is not without its flaws. Users must take special care that their `operator()` methods accept the correct arguments, which vary slightly between components. From a design standpoint, despite their relatively well-

separated behaviour, the functors structures are quite tightly coupled: each typically feeds information to the next via the per-ray data.

As they may contain state (e.g. pointers to global memory), functors are passed as function arguments to the kernel. This has the advantage that template type-deduction reduces the number of template arguments which must be explicitly provided to the traversal function; however, it also prohibits use of `virtual` methods and functors.[25]

Finally, the intersection processing and post-traversal ray-processing functors are likely to have significant side-effects, often writing data to global memory. In practice, all these factors combined make the behaviour of a set of GRACE functors difficult to interpret and debug, requiring every component to be understood in some detail. This is troublesome for code maintenance, where seemingly-innocuous changes to one component will break another.

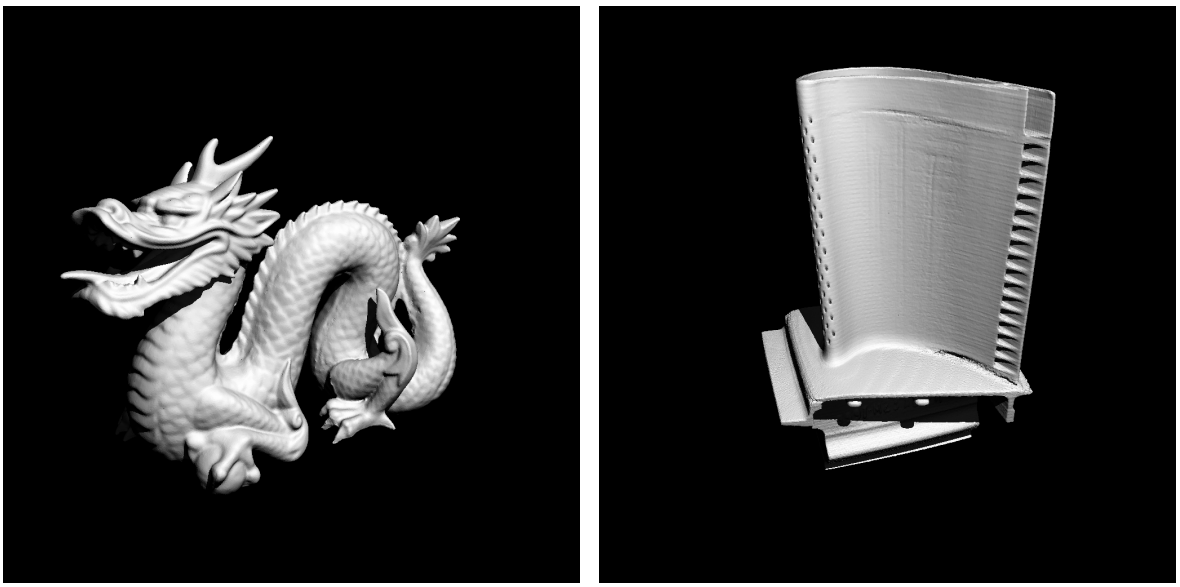
10.1 Application to triangle meshes

To demonstrate the use of the customizability available with GRACE, I have implemented a closest-intersection traversal function with triangles as the underlying primitive. The per-ray data contains the distance to the closest-found intersection, and is used within the intersection function as an effective ray length; intersected triangles beyond the current closest value return false (a miss). For ray-triangle intersection, the Möller-Trumbore (Möller and Trumbore, 2005) test is used.

This configuration has been applied to two triangle-mesh datasets typically used to benchmark ray tracing codes in the CG literature. A simple Lambertian reflectance model was assumed, implemented with flat shading (regardless of where on its surface a triangle is intersected, its colour and brightness due to a given light source is a constant). A pinhole camera model is assumed (and provided by GRACE as a hitherto-unmentioned method of ray generation). Shadow rays were also traced (rays which are emitted from the closest-found intersection of a camera ray and traced towards the light source(s); if a shadow ray finds an intersection, then its point of origin is assumed to be shaded). Results are shown in Figure 4.23.

A not-insubstantial amount of code is required in order to produce these images: a triangle class, triangle AABB and triangle centroid functors, ray-triangle intersection

[25] The CUDA runtime does support `virtual` inheritance and member functions on the device, but only if instances are constructed on the device.



(a) Stanford Dragon. **(b)** Turbine Blade.

Figure 4.23. Two common CG scenes rendered with GRACE, using a basic Lambertian flat shading model and shadow rays. Stanford dragon courtesy of the Stanford University Computer Graphics Laboratory; Turbine blade courtesy of the Georgia Institute of Technology Large Geometric Models Archive.

routine, pixel-shading kernel, light(s) and camera setup, shadow-ray generation kernel, and the GRACE pre-traversal, post-traversal, intersection test and intersection processing functors for both camera and shadow rays (though they are largely identical for both ray types). This code is, nonetheless, largely domain-specific, and one might expect quite trivial for those familiar with rendering.

10.2 Optimization of closest-intersection traversal

The implementation of a closest-intersection traversal kernel allows for a direct comparison of traversal performance to quoted literature values. Specifically, those of Aila et al. (2013), who present results for an LBVH acceleration structure across a number of scenes. They use the ray tracing kernels described in Aila and Laine (2009) and Aila et al. (2012), testing on a GTX 680 device. In keeping with their methods, GRACE builds a pure LBVH structure, using Morton keys for the ALBVH δ-function. As a caveat to such a comparison, Aila et al. (2013) trace 'diffuse inter-reflection' rays, which are incoherent, whereas here coherent rays, projected from a virtual pinhole-camera, are traced. This will be addressed later.

Note also that, while closest-intersection traversal is framed here as a computer

graphics use case, and hence already well-served by OptiX and other packages, the principles are applicable to radiative transfer and other scientific domains; for example, modelling scattering in optically thick media, or composing near-neighbour lists of particles, the latter being relevant for both SPH simulations and molecular dynamics.

As a test dataset, the Stanford Dragon was used, shown in Figure 4.23a. Results report the mean time to find the closest intersection for 1024^2 rays over ten iterations and four camera angles. A maximum leaf size of $\phi_{\max} = 8$ was initially chosen, being approximately optimal. With all (applicable) GRACE optimizations discussed previously, times listed as the 'Base' implementation in Table 4.16 give a starting point.

In GRACE, when a ray or packet intersects both child nodes of the current node, the left is traversed next, and the right is pushed to the stack. This decision is arbitrary and, until now, inconsequential. However, when searching for the closest intersection, it is advantageous to instead move to the closest child node first. Recall that the ray-AABB intersection test produces two values, $t_{\min}$ and $t_{\max}$ for the entry and exit points, respectively, along the ray; the 'closest' node is defined to be the one with the smaller $t_{\min}$ value. This is further complicated for packet traversal, where this definition will not necessarily result in agreement among threads in a warp. The implementation uses the `__popc()` and `__ballot()` intrinsics to count the number of threads which consider the right child node to be closer. Only if this number is greater than 16 (half the warp size) is the right node considered to be closer. An important additional factor is that the current closest-found intersection distance, t, be taken as the ray length in the ray-AABB intersection test. While only of little benefit ($\sim 3\%$ reduction in execution time) on its own, without it the closest-node-first traversal will visit exactly the same nodes as in the 'Base' implementation, albeit in a different order.

The final implementation, listed as 'Closest first' in Table 4.16, shows approximately a factor of two performance improvement.

Now, the 'Closed first' GTX 670 results of Table 4.16 are equivalent to 59.9M rays s^{-1}, while Aila et al. (2013) achieve 112M rays s^{-1}. Even accounting for the performance disparity between the GTX 670 and GTX 680 devices, clearly there is room for improvement. Examining the results of Aila and Laine (2009) and Aila et al. (2012), henceforth A09 and A12, respectively, we return to a point noted in Section 9.5, that those authors do not find packet tracing to achieve the highest performance. For their non-packet implementation, they go on to suggest 'speculative traversal' and 'persistent threads',

Table 4.16. Execution time for closest-hit packet traversal for the Stanford Dragon triangle-mesh dataset. Rays are coherent, simulating a pinhole camera. Given in brackets is the time as a fraction of the Base implementation.

Implementation	Execution time / ms	
	M2090	GTX 670
Base	54.2 (1×)	39.2 (1×)
Closest first	25.5 (0.47×)	17.5 (0.45×)

both of which aim to increase the SIMD efficiency by reducing execution divergence.

The former was already described in Section 7.3, referred to as 'postponed traversal'. In summary, an in-register stack variable is used to store the index of the next leaf to traverse, and said leaf can hence be postponed, allowing for extended traversal of inner nodes. This in turn increases the likelihood that all threads will find an intersected leaf and go on to do useful work when the warp switches to leaf traversal.

Persistent threads launches the kernel with a fixed number of threads, such that the device is filled (if possible), and warps then pick rays off the top of the input array, as needed; the top of the array is modified via an `atomicAdd()` operation to maintain consistency and prevent duplicated work. Threads exit when the array of input rays is exhausted. This system was initially proposed to work around the limitations of the thread scheduler in early CUDA devices. However, A09 also noted that terminated rays can be replaced as-needed on a per-thread basis, rather than by the warp as a whole, to further increase warp efficiency; this was fully realized in A12 with the advent of the aforementioned `__popc()` and `__ballot()` operations, which allow for an efficient implementation of the necessary warp-wide scan sum (see Listing C.1, page 251).

Before presenting these implementations, it may be helpful to examine the performance of non-packet traversal. Individual-ray traversal also prohibits the use of a shared memory cache for primitives (described in Section 7.3) in a leaf node. The impact of these changes on performance is given in the first three lines of Table 4.17, with a configuration otherwise identical to Table 4.16; the 'Packet' result in Table 4.17 is therefore identical to the 'Closest first' result in Table 4.16. Finally, for individual rays, a value of $\phi_{\max} = 2$ was chosen, being optimal.

Postponed-traversal, persistent-traversal, and a combination of both are also given in Table 4.17. Peak performance on a GTX 670 device is equivalent to 157M rays s^{-1},

Table 4.17. Mean execution time for closest-hit traversal under a number of schemes for the Stanford Dragon triangle-mesh dataset. 1024^2 coherent rays were traced, simulating a pinhole camera from four viewpoints. Given in brackets is the time as a fraction of the Base implementation of Table 4.16.

Implementation	Execution time / ms			
	M2090		GTX 670	
Packet	25.5	(0.47×)	17.5	(0.45×)
Packet no cache	25.7	(0.47×)	19.2	(0.49×)
Individual	15.4	(0.28×)	11.0	(0.28×)
Postpone	17.5	(0.32×)	11.3	(0.29×)
Persistent	9.06	(0.17×)	6.67	(0.17×)
Persistent-postpone	8.37	(0.15×)	7.11	(0.18×)

compared to the 112M *incoherent* rays s^{-1} reported by Aila et al. (2013) for LBVH on the higher-performance GTX 680.

Results are broadly consistent with the findings of A09 and A12. While removal of the shared memory cache for primitives has a minor negative impact on performance, this is more than offset by the gains made under an individual-ray implementation. Examining packet and individual ray implementations without this cache, but also without the closest-node-first optimization, individual rays increases runtime by $\sim 3\%$ on the M2090, and reduces runtime by $\sim 8\%$ on the GTX 670. Hence packets do not appear to perform uniformly or significantly worse when all-intersections traversal is performed on triangle-mesh datasets.

Unlike A12, here we see persistent threads to be equally favourable across both Fermi (M2090) and Kepler (GTX 670) architectures. Note that Fermi devices benefit from postponed leaf traversal only after the introduction of persistent threads. Finally, on Kepler devices, the postponed leaf scheme is harmful; this is consistent with A12, where it is only suggested for incoherent rays on this architecture.

The above results and conclusions hold when testing the turbine blade dataset, shown in Figure 4.23b, and fractional increases and decreases in runtimes are also similar. Of particular note for this dataset is the (GTX 670) runtime for the 'Base' (Table 4.16) implementation, of 71.2 ms, compared to the 'Persistent' (Table 4.17) implementation, of 4.74 ms. While the initial GRACE implementation results in a longer execution time relative to the smaller Stanford Dragon model, the optimized implementation

Table 4.18. Execution time for closest-hit traversal for both GRACE and the A09 and A12 kernels. For GRACE, the best results from Table 4.17 are reported. For the ALK results, the best results are reported (in all cases, the 'dynamic fetch' kernel, which is a persistent-postpone variant). Given in brackets is the time as a fraction of the GRACE implementation.

Implementation	Execution time / ms			
	Stanford Dragon		Turbine blade	
	M2090	GTX 670	M2090	GTX 670
GRACE	8.37	6.67	6.06	4.74
ALK	7.47 (0.89×)	6.17 (0.93×)	5.36 (0.88×)	4.41 (0.93×)

is actually faster, at 221M rays s^{-1} (Aila et al. (2013) also see a slight increase, to 120M *incoherent* rays s^{-1}). The optimizations applied to this point appear to be particularly effective for large, relatively flat and unvarying surfaces. This is despite a greater number of rays finding (at least one) intersection for the turbine blade (mean 2.5×10^5) than the dragon (mean 2.3×10^5).

Now, as has already been made clear, Aila et al. (2013) report results for incoherent rays, for which poorer performance is expected. However, the traversal kernels of A09 and A12 were made publicly available, henceforth the *ALK kernels*.[26] I have modified these kernels to use the Möller and Trumbore (2005) ray-triangle intersection routine used for GRACE, aiding a fair comparison. The ALK kernels are otherwise as-published. This required a (straightforward) conversion from the node `struct` used in GRACE, and, notably, the ALK kernels do not use an explicit leaf array. Instead, leaves are implicit: inner nodes with would-be leaf child nodes instead store the index of the first triangle in said leaf; triangles are stored contiguously, and the end of a leaf's set of triangles is marked by a sentinel value.

Performance, with an identical configuration to that used previously, is given in Table 4.18. In all cases the minimum-achieved runtime is reported, and again $\phi_{\max} = 2$ was found to be optimal. For GRACE, this is as in Table 4.17. For the ALK kernels, the 'dynamic fetch' kernel was fastest in all cases; it is a persistent-postpone variant, though, interestingly, denoted as optimal only for Kepler devices. (The ALK kernels do not include a persistent-only variant.)

GRACE is competitive with the ALK kernels. This is not surprising, as further

[26] `https://code.google.com/archive/p/understanding-the-efficiency-of-ray-traversal-on-gpus/` accessed 2016-11-14

Table 4.19. Mean execution time for closest-hit traversal under a number of schemes for a $z = 8$, $N = 128^3$ SPH dataset (n128-z8). 21 504 rays were traced, with randomly-generated origins and direction vectors. Given in brackets is the time as a fraction of the 'Base' GRACE implementation. An approximately-optimal leaf size of $\phi_{\max} = 20$ was used in all cases.

Implementation	Execution time / ms			
	M2090		GTX 670	
Base	100	(1.0×)	65.5	(1.0×)
Packet	54.7	(0.55×)	35.4	(0.54×)
Individual	14.7	(0.15×)	14.9	(0.23×)
Postpone	15.4	(0.15×)	15.4	(0.24×)
Persistent	13.0	(0.12×)	15.7	(0.24×)
Persistent-postpone	14.1	(0.14×)	15.5	(0.24×)
ALK	14.6	(0.15×)	13.0	(0.20×)

inspection shows that, at this point, they are quite similar. The primary differences lie in the exact use of the video min-max instructions (recall Section 7.3) in the ray-AABB intersection routine, logic for pushing and popping the traversal stack, ALK's use of the texture cache for both node and triangle fetches, and the aforementioned implicit leaf array. Further optimization for triangle-mesh datasets is therefore not pursued.

Moving back toward the scientific domain, the closest-hit kernel is applied to a set of randomly positioned and oriented rays in an SPH simulation. This is intended to approximate the workload of a photon-scattering simulation, though note that the same cosmological datasets of Section 9.2 are used, where scattering is typically considered a negligible effect. No ray sorting is performed, and hence they are highly incoherent. Results are given in Table 4.19, at optimization points discussed for Tables 4.16 to 4.18.

The closest-first optimizations thus far discussed are, broadly speaking, also useful for SPH datasets. Of note, the M2090 benefits more from said optimization than the GTX 670, and, surprisingly, the GTX 670 favours neither the persistent nor the postpone implementations. The use of incoherent rays seems to be the cause of these differences. Sorting rays by the Morton keys of their origins, we instead find the M2090 and GTX 670 both favour the persistent implementation. Further, while the 'Base' runtime is reduced to approximately 44 ms (0.66×) on the GTX 670, it is reduced to 55 ms (0.55×) on the M2090; the other implementations see much smaller gains after ray sorting. Thus it appears that the M2090 is rather less tolerant of incoherent-ray

workloads, but that the optimized implementations are not particularly sensitive to this incoherence.

11 Conclusion

In this chapter I have introduced GRACE, a high-performance ray tracing code designed primarily for cosmological smoothed particle hydrodynamics (SPH) simulations which runs on graphics processing units (GPUs).

Section 4 introduced the concept of parallel primitive operations, such as sorts and scan-sums, which can be performed on the GPU. Several high-performance libraries in the form of Thrust and MGPU were also noted, and my modified version of MGPU, SGPU. It was shown that, for the use-cases targeted by GRACE, SPU is approximately five times faster than Thrust.

The exact choice of acceleration structure, a bounding volume hierarchy (BVH), due to Apetrei (2014), was presented in Section 5. Substantial optimization of the implementation was carried out in Section 6. The time taken to group particles into leaves of sizes $\sim \phi_{\max} = 32$ was reduced by a factor of ten. A similar effort was then undertaken for the building of the full hierarchy, but was less successful, achieving at best gains on the order of a few percent. The performance of the implementation still falls short of that presented by other authors by a factor of a few. Fortunately, for the use-cases targeted by GRACE, tree hierarchy construction is not expected to be carried out frequently, and the performance of GRACE is more than sufficient.

Interpolation of the line-integral through an SPH particle from pre-computed integrals was found, in Section 7.2, to offer good accuracy. A novel linear interpolation method was also suggested, avoiding computation of a square root for each intersection but retaining accuracy.

In Section 7.3, significant effort to optimize BVH traversal was detailed. A compact array-of-structures (AoS) node layout and packet traversal were each found to reduce runtimes by up to $\sim 50\%$. A novel algorithm to improve coalescing when storing data at each intersection was also developed, which again reduced runtimes by approximately 50%. Several less significant or unsuccessful optimizations were also listed, and in particular, use of a higher-quality BVH was not found to be beneficial. Finally, the effectiveness of several ray-AABB intersection tests was examined, with simple, non-

branching implementations always being more effective on the GPU, in direct opposition to results found for the CPU.

The correct operation of the code was verified in Section 8 via comparison to a high-precision implementation and estimation of volume integrals over the SPH dataset, with results accurate to better than one part in 10^3.

Performance was presented in Section 9, along with comparisons to alternative libraries and codes. Despite earlier speculation to the contrary, the BVH construction implementation was shown to be competitive with the (presumably) well-optimized solution of OptiX, a ray tracing library developed by NVIDIA. GRACE also performs exceptionally well here relative to the octree builder in SPHRAY.

Moving on to ray tracing, GRACE was shown to achieve performance over an order of magnitude greater than that of SPHRAY and parallel CPU implementation, RTSPH, and a factor of a few greater than OptiX, when tracing $\gtrsim 10^4$ rays from a common origin. In Section 9.5 a more thorough investigation of the effectiveness of packet traversal was detailed. This confirmed that packet traversal is much less effective for lower per-source ray counts. In fact, its effectiveness has a somewhat non-trivial dependence on the typical size of a packet relative to the typical size of a particle. Reduced effectiveness can be mitigated by tracing rays from multiple sources simultaneously. Further, use of non-packet traversal scales well down to arbitrarily-low rays-per-source values, provided that the total number of rays being simultaneously traced is sufficient to fill the device. In any case, GRACE achieves substantially higher performance than researchers would typically achieve using their own CPU-based implementations, and for SPH datasets also compares favourably to a high-performance alternative, OptiX.

Finally, in Section 10, the user-configurable components of GRACE were described, and the substantial flexibility offered was demonstrated by application on triangle-meshes. GRACE allows for ray tracing datasets beyond SPH, with only a small amount of domain-specific code to be provided by users.

5

TARANIS: A GPU-accelerated numerical radiative transfer code

1 Introduction

TARANIS is a GPU-accelerated radiative transfer code. It is intended for cosmological SPH simulations, particularly of the Epoch of Reionization (EoR), and models radiation transport via ray tracing. The ray tracing component is supplied by GRACE.

TARANIS is a cumulative result of work by several authors, with contributions stated in Section 2. In Section 3, the parallel radiative transfer algorithm is presented, followed by a description of the implementation in Section 4. I have opted to test TARANIS by comparing its results to those of the first CRTCP (Iliev et al., 2006). These tests are presented in Section 5, and the analyses of the original project are repeated. Convergence and performance measurements are also performed for some of the CRTCP tests. Finally, in Section 6, I provide concluding remarks and suggestions for future work.

2 Contributions

The underlying numerical solver library, RT, has undergone several iterations. Initially a full radiative transfer code based on the algoritm of Abel et al. (1999), but extended to include helium chemistry, it was first presented by Bolton et al. (2004) and developed by James Bolton. A later version, worked on primarily by Eric Tittley, was introduced

in Tittley and Meiksin (2007), and that work serves as the primary reference for the RT code. This was ported to CUDA GPUs by Alex Bush for his Master's thesis, resulting in RT-CUDA, work which remains unpublished. Most recently (2015 – 2016), significant work was undertaken by Eric Tittley to generalize both RT and RT-CUDA to arbitrary volume elements, cuboidal cells having been previously assumed.

In their current form, RT and RT-CUDA are radiative transfer libraries which solve for ionization and temperature states of hydrogen-helium media, given the current state of a simulation element and the incident radiation (the relevant quantities are made more explicit in Section 4). These libraries are agnostic of the exact radiation-transport implementation.

The radiative transfer application, TARANIS, was primarily developed by Eric Tittley, in tandem with the above-noted generalizations of the RT libraries. It uses GRACE for ray tracing and RT-CUDA for evolution of the ionization and temperature state of a hydrogen-only medium. Where GRACE is used, it largely follows example code I have provided as part of that library. My other contributions to the TARANIS codebase are for bug fixes, features to aid usability, and general maintenance. Development of the parallel algorithm of Section 3 is the result of a collaborative effort between Eric Tittley and me, and builds on ideas of both Abel et al. (1999) and Altay et al. (2008) (SPHRAY).

I have performed all tests and analyses present in this chapter independently, unless otherwise stated.

3 A parallel radiative transfer algorithm

Much as for ray tracing, in order to achieve maximum performance it is essential that a significant majority of steps in a radiative transfer application be parallelized. Processing of ray-particle intersections and particle updates are the main targets for such an algorithm.

For the former, consider that for a given ray and particle, two pieces of information must be computed: the column density up to the current particle, which determines the amount of incident radiation, and the column density through the particle, which determines the fraction of that radiation which is absorbed. By design, GRACE makes this functionality available, proceeding as follows:

1. Trace all rays and store particle integrals, along-ray distance and particle index for all intersections.
2. Sort per-ray intersections by distance along the ray.
3. Scan-sum the particle integrals for each ray.

(Here, distance to an intersection is defined as the distance along the ray to the point of closest approach to a particle's centre; this is somewhat arbitrary, but is only significant at scales on the order of the smoothing length.) Thus, we obtain the cumulative particle integral (a proxy for column density) up to all intersected particles; the per-particle integrals are easily recovered as the difference between consecutive values in this cumulative array. A dummy value is added to the end of this array, storing the total integral along the ray, which allows the final particle's integral to be computed as a difference.

The method, thus far, is entirely compatible with SPHRAY, and likely suitable for parallelizing elements of other radiative transfer codes. It also somewhat alleviates the oversampling of close-to-source particles inherent in long characteristics ray tracing (recall Chapter 3, Section 3.4, page 51). Naïvely, one would trace a ray to each particle, compute the cumulative column density to that particle, and use this to estimate the incident radiant energy. Computational effort is then duplicated close to the source, where particles are intersected by many rays. It was observed by Abel et al. (1999) that, if all points of intersection along a ray are considered to affect their corresponding particles, then we need only trace rays to the outer shell of simulation elements; particles closer to the source(s) are likely to be intersected by at least one of these rays. In practice, to sample all N particles, only $N^{2/3}$ rays need be traced. On average, we expect rays which exit the volume to intersect $N^{1/3}$ particles, hence MN scaling, where M is the number of sources.

To parallelize particle-state updates, the contributions from each ray intersecting a given particle must be combined. The key point here is that this may be achieved by accumulating the ionization and heating rates due to each such ray, and performing the update once the total rate is known for every particle. These rates then also inform and constrain the size of the timestep. In this scheme, from the perspective of a particle, rays from the same sources and from different sources are treated identically.

4 Radiative transfer implementation

The numerical solver developed for SPHRAY, which evolves the ionization and temperature state of a particle, is not compatible with the above parallel algorithm. There, the number of photons deposited in a particle is allowed to vary as a particle's state is integrated over a timestep, and thus modifies the number of incident photons for the next particle along the ray. This is a particular problem where multiple rays intersect the same particle simultaneously (not possible in the serial SPHRAY code), as only one ray may update the particle state at a time. This is the primary cause of poor scaling in P-SPHRAY (recall Chapter 2, Section 4.3, page 34) (Yu Feng, private communication, 2012-12-06).

The RT solver behaves more like a traditional long characteristics solver, determining absorbed radiant energy based only on that which is incident and a particle's state at the beginning of the timestep. This is compatible with simultaneous particle updates, the trade-off being that the smallest-acceptable timestep is enforced for all particles in an update, despite (potentially) only being necessary for a single particle.

4.1 Ionization state

In Chapter 2, Section 2.1, radiative transfer was framed in terms of specific intensity as a function of position and time. However, in the context of cosmological radiative transfer simulations, it is convenient to begin with the equations governing the particle states we wish to evolve; namely, the ionization state of volume elements (temperature is discussed in Section 4.4). The RT libraries work in terms of the number densities of hydrogen and helium species,

$$\dot{n}_{\mathrm{HI}} = -n_{\mathrm{HI}}\,\Gamma_{\mathrm{HI}} + n_e\,n_{\mathrm{HII}}\,\alpha_{\mathrm{HII}}, \tag{5.1a}$$

$$\dot{n}_{\mathrm{HII}} = -\dot{n}_{\mathrm{HI}}, \tag{5.1b}$$

$$\dot{n}_{\mathrm{HeI}} = -n_{\mathrm{HeI}}\,\Gamma_{\mathrm{HeI}} + n_e\,n_{\mathrm{HeII}}\,\alpha_{\mathrm{HeII}}, \tag{5.1c}$$

$$\dot{n}_{\mathrm{HeII}} = -(\dot{n}_{\mathrm{HeI}} + \dot{n}_{\mathrm{HeIII}}), \tag{5.1d}$$

$$\dot{n}_{\mathrm{HeIII}} = n_{\mathrm{HeII}}\,\Gamma_{\mathrm{HeII}} - n_e\,n_{\mathrm{HeIII}}\,\alpha_{\mathrm{HeIII}}, \tag{5.1e}$$

where $\dot{n}$ denotes a derivative with respect to time, n_A is the number density of species A, n_e the electron number density, Γ_A (s^{-1}) the photoionization rate per atom of species A,

Table 5.1. Recombination coefficients.

Coefficient (m^3/s)	Fit
α^A_{HII}	$2.065 \times 10^{-17}\, T^{-1/2} \left(6.414 - \frac{1}{2}\ln T + 8.68 \times 10^{-3}\, T^{1/3}\right)$
α^A_{HeII}	$1.27 \times 10^{-16}\, T^{-0.5526} \exp\left(-6.875 \times 10^{-3}\, \ln T \ln T\right)$
η^A_{HeII}	$1.9 \times 10^{-9} \left[1 + 0.3 \exp\left(\frac{-9.4\times10^4}{T}\right)\right] \exp\left(\frac{-4.7\times10^5}{T}\right) T^{-3/2}$
$\alpha^A_{\mathrm{HeIII}}$	$8.260 \times 10^{-17}\, T^{-1/2} \left(7.107 - \frac{1}{2}\ln T + 5.47 \times 10^{-3}\, T^{1/3}\right)$
α^B_{HII}	$2.065 \times 10^{-17}\, T^{-1/2} \left(5.620 - \frac{1}{2}\ln T + 8.68 \times 10^{-3}\, T^{1/3} + 2.01 \times 10^{-5}\, T^{0.8}\right)$
α^B_{HeII}	$1.007 \times 10^{-16}\, T^{-0.5095} \exp\left(-1.429 \times 10^{-2}\, \ln T \ln T\right)$
η^B_{HeII}	$1.9 \times 10^{-9} \left[1 + 0.3 \exp\left(\frac{-9.4\times10^4}{T}\right)\right] \exp\left(\frac{-4.7\times10^5}{T}\right) T^{-3/2}$
$\alpha^B_{\mathrm{HeIII}}$	$8.260 \times 10^{-17}\, T^{-1/2} \left(6.320 - \frac{1}{2}\ln T + 5.47 \times 10^{-3}\, T^{1/3} + 6.64 \times 10^{-3}\, T^{0.8}\right)$

and α_A the recombination coefficient (m^3/s) from species A, a function of temperature. The ionization fractions are also convenient quantities, $x_{\mathrm{HI}} \equiv n_{\mathrm{HI}}/n_{\mathrm{H}}$ and likewise for x_{HII}, x_{HeI}, x_{HeII} x_{HeIII}. Case A and case B (the on-the-spot approximation, see Chapter 2, Section 2.2) recombination coefficients are supported by RT. Both α_{HII} and α_{HeIII} values are from Seaton (1959); the HeII coefficient is composed of radiative (α) and dielectric (η) components, the former a fit by James Bolton to Hummer and Storey (1998), the latter from Aldrovandi and Péquignot (1973).

All recombination coefficients are given in Table 5.1; T is in units of Kelvin. These are straightforward to implement, and Eqs (5.1) are ultimately solved via a simple forward Euler integration of the form

$$n(t + \Delta t) = n(t) + \dot{n}(t)\, \Delta t.$$

It still remains, however, to compute the photoionization rate, which is discussed in the following sections.

4.2 Optical depth

Recall the time-independent (constant absorption and emission) three-dimensional equation of radiative transfer, Eq. (2.18), page 14. In ray tracing, the three spatial dimensions are sampled along one-dimensional rays, and so we reduce this to its one-dimensional form, Eq. (2.7), page 10. Finally, the source term is considered to be zero, except at defined source points, hence we are to solve Eq. (2.8). Then the specific

intensity at a point s along a ray is that of Eq. (2.10),

$$I_\nu(s) = I_{\nu,0}\, e^{-\tau_\nu(s)}. \tag{5.2}$$

where $I_{\nu,0}$ is the source intensity and τ_ν is as in Eq. (2.12),

$$\tau_\nu(s) = \int_0^s \alpha_\nu(s')\ \mathrm{d}s', \tag{5.3}$$

and $\alpha_\nu = n\sigma_\nu$ accounts for all (modelled) photoabsorptions, where σ_ν is the cross-section and n the number density of photoabsorbers.

Now, recall that GRACE computes the cumulative SPH kernel integral along a ray. This is converted to a proxy for column density for each photoabsorbing species of interest,

$$N_i^{\mathrm{col}} = \mathcal{I}_i\, x_i\, N_i, \tag{5.4}$$

where $\mathcal{I}_i$ is the cumulative integral for the ith intersection, x_i is the ionization fraction for the ith intersected particle, and N_i is the number of atoms of the photoabsorbing species per simulation volume element (a constant for constant-mass elements, as is common in SPH). The local, or per-particle, column density proxy for the ith intersected particle is easily recovered,

$$\Delta N_i^{\mathrm{col}} = N_{i+1}^{\mathrm{col}} - N_i^{\mathrm{col}}. \tag{5.5}$$

The cumulative optical depth to a particle, for a given species and at a given frequency ν, is therefore computed as

$$\tau_\nu = \sigma_\nu\, N^{\mathrm{col}}, \tag{5.6}$$

and the optical depth through the particle is

$$\Delta\tau_\nu = \sigma_\nu\, \Delta N^{\mathrm{col}}, \tag{5.7}$$

where in both cases the indices i have been elided for brevity. In the case of multiple photoabsorbing species, total cumulative and local optical depths are computed simply as the sum over the per-species depths, henceforth $\sum \tau_\nu$ and $\sum \Delta\tau_\nu$ respectively.

Table 5.2. Fits to Eq. (5.9) from Osterbrock (1989), pg. 36, for hydrogen and helium.

Ionization	v_0 / Hz	σ_{ν_0} / m^2	β	s
HI $\rightarrow$ HII	3.289×10^{15}	6.30×10^{-22}	1.34	2.99
HeI $\rightarrow$ HeII	5.954×10^{15}	7.83×10^{-22}	1.66	2.05
HeII $\rightarrow$ HeIII	1.316×10^{16}	1.58×10^{-22}	1.34	2.99

4.3 Photoionization rates

Recalling Chapter 2, Section 2.4, the photoionization rate per unit volume for a given species as a function of the local (angular-)mean specific intensity of radiation, $4\pi J_\nu \equiv \int_\Omega I_\nu$, and the ionization cross-section, σ_ν, is

$$n\Gamma = 4\pi \int_{\nu_0}^{\infty} \frac{J_\nu}{h\nu} n\sigma_\nu \, \mathrm{d}\nu, \tag{5.8}$$

where ν_0 is the species-dependent ionization threshold and h is Planck's constant.

RT uses the cross-sections of Osterbrock (1989),

$$\sigma(\nu) = \sigma_{\nu_0} \left[\beta \left(\frac{\nu}{\nu_0} \right)^{-s} + (1-\beta) \left(\frac{\nu}{\nu_0} \right)^{-(s+1)} \right], \tag{5.9}$$

with values tabulated for each species in Table 5.2.

The mean intensity at a particle is estimated by the ray tracing procedure, and is a function of the spectral luminosity of a source, L_ν (the power per unit frequency), and the cumulative optical depth to the particle, τ_ν.

In general, the mean intensity a distance r from a source is

$$4\pi J_\nu(r) = f(r) L_\nu \equiv S_\nu(r),$$

where f is an attenuation factor with units of inverse area, and we have introduced the spectral flux density, S_ν (power per unit frequency per unit area).

In cases where ray tracing is used only to estimate S_ν, a distance r from a source we have

$$S_\nu(r) = \frac{L_\nu \exp\left(-\sum \tau_\nu\right)}{4\pi r^2} \tag{5.10}$$

where both geometric attenuation and attenuation due to intervening matter are accounted for. In TARANIS, however, rays are considered to transport photons directly.

For a given source, we consider each ray to represent a specific luminosity $L_\nu/N_{\rm rays}$ at emission. Physically, this corresponds to a power per unit frequency in a given solid angle, the solid angle being $\Omega = 4\pi/N_{\rm rays}$ for isotropically distributed rays; that is, a ray approximates a beam. Now, consider that a ray-element intersection induces a fixed ionization rate across its entire volume. Provided that $\Omega\, r \lesssim A$, where A is the cross-sectional area of the volume element, we instead find

$$S_\nu(r) = \frac{L_\nu \exp\left(-\sum \tau_\nu\right)}{A N_{\rm rays}}, \tag{5.11}$$

where attenuation due to intervening matter is unchanged from above.

For $\Omega\, r < A$, oversampling occurs, and the approximated beam is incident only over a small fraction of the simulation element's cross section. Nonetheless, it is by necessity considered to irradiate the entire element, and the factor $1/A$ remains appropriate. For $\Omega\, r > A$, instead the beam should irradiate more simulation elements than are actually intersected, and the ray tracing result will be in error.

Finally, note that, for a point source, the number density of isotropically-distributed rays intersecting a fixed area a distance r from the source is $\propto 1/4\pi r^2$, and indeed there should be no such term in Eq. (5.11). This applies also for extremely distant sources modelled with plane-parallel rays, where there is no such intrinsic reduction in the number density of rays per unit area. If one assumes that the particle-to-source distance is much greater than the simulation box side-length, $r \gg L$, then in fact there is negligible geometric attenuation.

In terms of the convenient S_ν, the ionization rate is

$$n\Gamma = \int_{\nu_0}^{\infty} \frac{S_\nu \exp\left(-\sum \tau_\nu\right)}{h\nu}\, n\sigma_\nu \,\mathrm{d}\nu. \tag{5.12}$$

This is problematic as it is valid only at some specific (time and) distance r from the source, while a rate valid over some Δr across the simulation volume element is required. Taking Eq. (5.12) as a constant across an element does not guarantee photon conservation: the number of ionizations produced by a ray will not necessarily be consistent with the energy absorbed along that ray, the latter inferred via the value of τ_ν. Abel et al. (1999) noted this problem, and suggested that ionization rates be forced to match absorption rates. While their formalism considers packets of photons and the number of incident ionizing photons per unit time, we may equivalently apply the idea

here.

Consider the rates at two points, $n\Gamma(r)$ and $n\Gamma(r+\Delta r)$, where Δr is on the order of the simulation element scale. Now, define the spatially-averaged rate over this region as

$$\langle n\Gamma \rangle = \frac{1}{\Delta r} \int_r^{r+\Delta r} n\Gamma(s)\,\mathrm{d}s. \tag{5.13}$$

If we assume $\Delta r \ll r$, then we may take $S_\nu(r) \simeq S_\nu(r+\Delta r)$. By definition, n and σ_ν are taken to be constant across a volume element, which also implies $\tau_\nu(s) = n\sigma_\nu s$ for $r \leq s \leq r+\Delta r$. Thus,

$$\begin{aligned}
\langle n\Gamma \rangle &= \frac{1}{\Delta r} \int_{\nu_0}^{\infty} \frac{S_\nu}{h\nu}\, n\sigma_\nu \int_r^{r+\Delta r} \exp\left(-\sum n\sigma_\nu s\right)\,\mathrm{d}s\,\mathrm{d}\nu \\
&= \frac{1}{\Delta r} \int_{\nu_0}^{\infty} \frac{S_\nu}{h\nu} \left[-\exp\left(-\sum n\sigma_\nu s\right)\right]_r^{r+\Delta r} \mathrm{d}\nu \\
&= \frac{1}{\Delta r} \int_{\nu_0}^{\infty} \frac{S_\nu}{h\nu} \exp\left[-\sum \tau_\nu(r)\right] \left\{1 - \exp\left[-\sum \tau_\nu(\Delta r)\right]\right\} \mathrm{d}\nu.
\end{aligned}$$

In the notation used previously,

$$\langle n\Gamma \rangle = \frac{1}{\Delta r} \int_{\nu_0}^{\infty} \frac{S_\nu \exp\left(-\sum \tau_\nu\right)}{h\nu} \left[1 - \exp\left(-\sum \Delta\tau_\nu\right)\right] \mathrm{d}\nu, \tag{5.14}$$

where $\tau_\nu = \tau_\nu(r)$ and $\Delta\tau_\nu = \tau_\nu(\Delta r)$. One will note that we recover Eq. (5.12) if we take $1 - \exp\left(-\sum \Delta\tau_\nu\right) \approx \Delta\tau_\nu = n\sigma_\nu \Delta r$, which indeed holds in the locally optically thin limit, $\sum \Delta\tau_\nu << 1$. Comparing to Eq. (5.12), the left-hand term in the integrand of Eq. (5.14) is unchanged, simply expressing the incident spectral flux density. The new right-hand term is interpreted as an absorption probability, i.e. the fraction of incident photons which are absorbed. This rate is also now clearly consistent.

To convert the above to a per-species photoionization rate, we weight the absorption factor, f, defined as

$$f = 1 - \exp\left(-\sum \Delta\tau_\nu\right). \tag{5.15}$$

The absorption factor for a particular species, f_A, is then

$$f_A = \frac{\Delta\tau_\nu^A}{\sum_{A'} \Delta\tau_\nu^{A'}}\, f, \tag{5.16}$$

where $\Delta\tau_\nu^A$ is the local optical depth for species A. This replaces the factor f in Eq. (5.14) and we obtain a per-species ionization rate,

$$\langle n_A \Gamma_A \rangle = \frac{1}{\Delta r} \int_{\nu_0}^{\infty} \frac{S_\nu \exp\left(-\sum \tau_\nu\right)}{h\nu} \frac{\Delta \tau_\nu^A}{\sum \Delta \tau_\nu} \left[1 - \exp\left(-\sum \Delta \tau_\nu\right)\right] \mathrm{d}\nu, \tag{5.17}$$

where as before the sums are over all species. For a frequency $\nu < \nu_0^A$, where ν_0^A is the ionization threshold for species A, the local optical depth is taken to be zero.

As an implementation detail, note that f_A contains the term

$$\frac{1 - \exp\left(-\sum \Delta \tau_\nu\right)}{\sum \Delta \tau_\nu},$$

which for efficiency is approximated as $1/\sum \Delta \tau_\nu$ and $1 - \sum \Delta \tau_\nu / 2$ for large[1] and small[2] $\sum \Delta \tau_\nu$, respectively.

It now remains to compute the integral and, perhaps more subtly, determine a definition of Δr appropriate for SPH. For the latter, we adopt

$$\Delta r \equiv \frac{4}{3} R \equiv \frac{4}{3} \left(\frac{3m}{4\pi\rho}\right)^{1/3}, \tag{5.18}$$

where m and ρ are the mass and density of the SPH particle in question. R is an effective radius, and physically analogous to the side-length of a cell in a grid-based hydrodynamics solver; $4R/3$ is the mean path-length through a sphere of radius R. (The definition $R = h/N_{\mathrm{ngb}}^{1/3}$ also seems appropriate, but is inaccurate for datasets where N_{ngb} is not constant, as was (unexpectedly) observed in some GADGET-2 outputs.) As an aside, note that it is tempting, but incorrect, to associate Δr with the per-intersection path-length through a simulation element. Above, we derived $\langle n\Gamma \rangle$ by spatially-averaging over the scale of interest, Δr. Regardless of the path-length through a cell or particle, the rate applies to the entire element, and hence the scale is fixed. Rather, the absorption-probability term accounts for differing path-lengths, being a function of $\Delta \tau_\nu$. The cross-section in Eq. (5.11) is simply $A = \pi R^2$.

The integral Eq. (5.17) is separated over three frequency ranges, $[\nu_0^{\mathrm{HI}}, \nu_0^{\mathrm{HeI}})$, $[\nu_0^{\mathrm{HeI}}, \nu_0^{\mathrm{HeII}})$ and $[\nu_0^{\mathrm{HeII}}, \infty)$. The first two are estimated numerically by Gauss-Legendre quadrature, and the third by Gauss-Laguerre.

Gauss-Legendre quadrature provides the following approximation to the definite integral of a function $f(x)$, and is exact for all polynomials f up to degree $2n - 1$:

[1] Values $> p^{-1}$ where p is the ISO C standard machine epsilon, which depends on whether RT is compiled for single- or double-precision.

[2] Values < 0.01, resulting in errors $\lesssim 10^{-5}$

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \simeq \sum_{i=1}^{n} w_i f(x_i), \tag{5.19}$$

with the weights

$$w_i = \frac{2}{(1-x_i^2)\left[P_n'(x_i)\right]^2}, \tag{5.20}$$

where P_n' is the first derivative of the nth Legendre polynomial, and the abscissas x_i are the roots of P_n. A straightforward linear transformation of Eq. (5.19) yields the more useful

$$\int_a^b f(x)\,\mathrm{d}x \simeq \frac{b-a}{2}\sum_{i=1}^{n} w_i\, f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right). \tag{5.21}$$

Gauss-Laguerre quadrature instead approximates

$$\int_0^\infty e^{-x} f(x)\,\mathrm{d}x \simeq \sum_{i=1}^{n} w_i f(x_i), \tag{5.22}$$

with weights

$$w_i = \frac{1}{x_i\left[L_n'(x_i)\right]^2}, \tag{5.23}$$

where L_n' is the first derivative of the nth Laguerre polynomial, and the abscissas x_i are the roots of L_n. It is straightforward to derive the more useful

$$\int_a^\infty g(x)\,\mathrm{d}x \simeq a\sum_{i=1}^{n} w_i e^{x_i} g\left[a(x_i+1)\right]. \tag{5.24}$$

The forms Eqs (5.21) and (5.24) are used to integrate Eq. (5.17) over the above-noted ranges, with $n = 4$ for Gauss-Legendre and $n = 8$ for Gauss-Laguerre; weights (w_i) and abscissas (x_i) are given in Table 5.3. For efficiency, the photoionization cross-sections of Eq. (5.9) are pre-computed for all required sample frequencies $\nu_i(x_i)$.

In many cases a particle will be intersected by multiple rays. All intersections are processed in parallel, and result in an ionization rate contribution for the intersected particle. Once computed, these contributions are summed for each particle in a following stage via an `atomicAdd()` operation. Thus we obtain a photoionization rate for each particle. This extends trivially to multiple sources. In TARANIS, each source is looped over in turn, with rates accumulated at the end of each such loop.

4.4 Photoheating and cooling

RT follows the thermal state of the gas via the entropic function $A(S)$, defined by

Table 5.3. Abscissas and weights for Gauss-Legendre and Gauss-Laguerre quadrature. Weights given for Gauss-Laguerre are the values of $w_i e^{x_i}$.

Method	x_i	w_i
Gauss-Legendre	−0.861 136 311 594	0.347 854 845 137
	−0.339 981 043 585	0.652 145 154 863
	0.339 981 043 585	0.652 145 154 863
	0.861 136 311 594	0.347 854 845 137
Gauss-Laguerre	0.170 279 632 305	0.437 723 410 493
	0.903 701 776 799	1.033 869 347 67
	2.251 086 629 87	1.669 709 765 66
	4.266 700 170 29	2.376 924 701 76
	7.045 905 402 39	3.208 540 913 35
	10.758 516 010 2	4.268 575 510 83
	15.740 678 641 3	5.818 083 368 67
	22.863 131 736 9	8.906 226 215 29

$$P = A(S)\rho^{\gamma} \tag{5.25}$$

where P is pressure, ρ is density, γ is the adiabatic index and S is the specific entropy, which we define as

$$S \equiv k_B(\gamma - 1)^{-1} \ln\left(P\rho^{-\gamma}\right), \tag{5.26}$$

where k_B is the Boltzmann constant. Combining the above with the ideal gas law $P = \frac{k_B}{\mu m_u}\rho T$, where μ is the mean molecular weight, m_u the atomic mass unit, and T the temperature; the second law of thermodynamics, $\frac{\mathrm{d}S}{\mathrm{d}t} = \frac{1}{T}\frac{\mathrm{d}Q}{\mathrm{d}t}$, where Q is energy transfer to the system; and the relation $n\mu m_u = \rho$, where $n = n_{\mathrm{H}} + n_{\mathrm{He}} + n_e$ is the total number density, we arrive at

$$\frac{\mathrm{d}A}{\mathrm{d}t} = (\gamma - 1)\rho^{-\gamma}(G - L). \tag{5.27}$$

G and L represent heating and cooling per unit volume per unit time, respectively. Temperature is recovered as

$$T = \frac{\mu m_u}{k_B} A\rho^{\gamma - 1} \tag{5.28}$$

.

Photoheating is due entirely to excess energy left over after ionization. Thus, from

Eq. (5.12)

$$G = \int_{\nu_0}^{\infty} \frac{S_\nu \exp\left(-\sum \tau_\nu\right)}{\nu} \left(\nu - \nu_0\right) n\sigma_\nu \, \mathrm{d}\nu. \tag{5.29}$$

And again in analogy with Eq. (5.14),

$$\langle G \rangle = \frac{1}{\Delta r} \int_{\nu_0}^{\infty} \frac{S_\nu \exp\left(-\sum \tau_\nu\right)}{\nu} (\nu - \nu_0) \left[1 - \exp\left(-\sum \Delta\tau_\nu\right)\right] \mathrm{d}\nu. \tag{5.30}$$

As an implementation detail, per-species heating rates $\langle G_A \rangle$ are computed in a manner identical to Eq. (5.17), and summed to give the total rate above. This is in order that the various identical components within the integrands of G and $n\Gamma$ may be calculated only once for each sample frequency ν_i.

Similarly to the per-intersection photoionization rates, multiple intersections may produce a photoheating rate for the same particle. Again, these rates are accumulated for each particle via an `atomicAdd()` operation.

RT considers cooling due to recombinations (L_{HII}, L_{HeII} and L_{HeIII}), collisional excitations of neutral hydrogen (L_e), and inverse Compton scattering off CMB photons (L_C), for a total cooling rate

$$L = L_{\mathrm{HII}} + L_{\mathrm{HeII}} + L_{\mathrm{HeIII}} + L_e + L_C. \tag{5.31}$$

These values are computed on a per-particle basis, not per-intersection.

For a species A, recombinations radiate the electron energy at a rate $L_A = n_e n_A \beta_A(T)$, where n_e is the electron number density and $\beta_A(T)$ is the recombination cooling coefficient, a function of temperature. Case A and case B (the on-the-spot approximation, see Chapter 2, Section 2.2) recombination coefficients are supported by RT. Both β_{HII} and β_{HeIII} fits are from Hui and Gnedin (1997); the HeII coefficient is a sum of radiative (β) and dielectric components, the former a fit by James Bolton to Hummer and Storey (1998), the latter equal to $3\,\mathrm{Ryd}\,\eta_{\mathrm{HeII}}$ as per Gould and Thakur (1970), where the η_{HeII} coefficient is the dielectric term in Table 5.1. All other recombination cooling coefficients are given in Table 5.4; T is in units of Kelvin.

For the cooling rate from collisional excitation of neutral hydrogen, the fits of Scholz and Walters (1991) are used,

Table 5.4. Recombination cooling coefficients.

Coefficient (m^3/s)	Fit
β^A_{HII}	$1.137 \times 10^{-31}\, T^{-0.965} \left(1 + 784.4\, T^{-0.502}\right)^{-2.697}$
β^A_{HeII}	$2.80 \times 10^{-39}\, \frac{T^{-0.5971}}{1+\left(\frac{T}{2.29}\right)^{0.3037}}$
β^A_{HeIII}	$3.466 \times 10^{-30}\, T^{-0.965} \left(1 + 1573\, T^{-0.502}\right)^{-2.697}$
β^B_{HII}	$2.340 \times 10^{-32}\, T^{-0.970} \left(1 + 86.15\, T^{-0.376}\right)^{-3.720}$
β^B_{HeII}	$1.53 \times 10^{-39}\, \frac{T^{-0.4020}}{1+\left(\frac{T}{5620}\right)^{0.4440}}$
β^B_{HeIII}	$7.183 \times 10^{-31}\, T^{-0.970} \left(1 + 145.1\, T^{-0.376}\right)^{-3.720}$

Table 5.5. Collisional excitation cooling fits.

Coefficient	Fit	
	$2 \times 10^3\,\mathrm{K} < T \leq 10^5\,\mathrm{K}$	$10^5\,\mathrm{K} < T < 1 \times 10^8\,\mathrm{K}$
d_0	$2.137\,913 \times 10^2$	$2.712\,544\,6 \times 10^2$
d_1	$-1.139\,492 \times 10^2$	$-9.801\,945\,5 \times 10^1$
d_2	$2.506\,062 \times 10^1$	$1.400\,727\,6 \times 10^1$
d_3	$-2.762\,755$	$-9.780\,842\,1 \times 10^{-1}$
d_4	$1.515\,352 \times 10^{-1}$	$3.356\,289\,1 \times 10^{-2}$
d_5	$-3.290\,382 \times 10^{-3}$	$-4.553\,323\,1 \times 10^{-4}$

$$L_e = 1 \times 10^{-33} \exp\Big[d_0 + d_1 \ln T + d_2 (\ln T)^2 + d_3 (\ln T)^3 + d_4 (\ln T)^4 + d_5 (\ln T)^5 - 1.184\,156 \times 10^5\, T^{-1}\Big], \tag{5.32}$$

where the coefficients d_i are given in Table 5.5.

Finally, inverse Compton scattering (e.g. Peebles, 1968),

$$L_C = \frac{4\sigma_T a k_B}{m_e c} \left[T_{\mathrm{CMB}}(1+z)\right]^4 (T - T_{\mathrm{CMB}})\, n_e, \tag{5.33}$$

where σ_T is the Thomson cross-section, a is the radiation energy density constant, m_e is the electron mass, c is the speed of light, T_{CMB} is the current temperature of the cosmic microwave background, and z is the current redshift.

4.5 Euler integration timestep

As noted in Section 4.1, quantities are updated via a forward Euler integration over a timestep Δt. At each such step, this timestep is applied equally to all particles and all quantities. The timestep is therefore constrained by the minimum-allowed value of a particular property of a particular particle. Before continuing, note that while helium has generally been considered above, and is supported by RT, TARANIS currently only considers ionization and photoheating of hydrogen, and therefore timescales associated with helium are ignored. In any case, the principles below are easily extended to multiple species.

Entropy (τ_s) and ionization (τ_n) timescales are computed as

$$\tau_s = \left| \frac{A}{(\gamma - 1)\rho^{-\gamma}(G - L)} \right|, \tag{5.34a}$$

$$\tau_n = \begin{cases} \dfrac{-n_{\mathrm{HI}}}{\dot{n}_{\mathrm{HI}}}, & \text{if } \dot{n}_{\mathrm{HI}} < 0 \\ \dfrac{-n_{\mathrm{HII}}}{\dot{n}_{\mathrm{HII}}}, & \text{if } \dot{n}_{\mathrm{HI}} > 0 \end{cases}, \tag{5.34b}$$

where terms are as defined in Section 4.1 and Section 4.4. For each particle i, the minimum timescale $\tau^i = \min(\tau_s^i, \tau_n^i)$ is computed, and the global minimum $\tau = \min\{\tau^0, \tau^1, ..., \tau^n\}$. The latter is implemented via a call to `thrust::reduce()`. Finally, $\Delta t = f \cdot \tau$, where typically $f \lesssim 1$. Here, $f = 0.2$ is used unless otherwise stated.[3]

5 The cosmological radiative transfer comparison project

With the implementation covered, we now move on to testing. All but the very simplest (see Section 5.1) radiative transfer problems are analytically intractable, hence the development of radiative transfer codes. However, this presents a challenge when verifying the correct operation of the software — to what should its results be compared? One possible answer is to compare the output of several different such applications, given the same initial conditions. This is the approach taken by the Cosmological Radiative Transfer Comparison Project (CRTCP), first published as Iliev et al. (2006), henceforth IL06, and has been followed by other authors (e.g. Altay et al., 2008; Pawlik

[3] This choice is somewhat arbitrary, and will later be verified; however, the RT library has previously shown convergence for $f < 0.5$.

and Schaye, 2011). To the same end, I have run the static-field tests with TARANIS. For consistency in the analysis, I have also re-analysed all (available) CRTCP data.

Note that astrophysical codes tend to evolve on faster timescales than the publications describing them (at least, those which are actively maintained); further, IL06 is now over a decade old. Ideally, then, TARANIS would be compared to the most-recent versions of codes participating in the CRTCP, and those which have been developed since. Unfortunately, such an undertaking is beyond the scope of this thesis. However, the current version of SPHRAY (Altay et al., 2008, henceforth AL08) has been used to re-run all tests. Since SPHRAY operates directly on SPH datasets, this also allows for comparisons with identical input datasets. Unless otherwise stated, a total of 10^8 rays are traced with SPHRAY; I have confirmed that this results in convergence, with results essentially indistinguishable from runs with a total of 10^9 rays. Convergence tests for TARANIS are detailed in Section 5.5.

The accompanying paper for the CRTCP, IL06, serves as a reference for both the initial conditions of each test and the following analyses. In general, results of IL06 are reproduced here; however, several discrepancies are noted in Appendix E. Much of the test data is, at time of writing, still available.[4] However, all data for the ART code is unavailable, and both OTVET and ZEUS data are unavailable for Test 3.

In all tests, case B (on-the-spot approximation) recombination and cooling rates are used.

Finally, as stated, all analysis has been redone for all available CRTCP data. Line colours and styles have, as far as reasonably possible, been matched to those of IL06; nonetheless, a key is provided below. C^2-RAY (Mellema et al., 2006), FLASH hybrid characteristics (Rijkhorst et al., 2006), FTTE (Razoumov and Cardall, 2005), RSPH (Susa, 2006) and ZEUS (Whalen and Norman, 2006) and all solve non-equilibrium chemistry equations and estimate optical depth by some form of ray tracing, so are the most relevant comparisons besides SPHRAY. CRASH (Maselli et al., 2003) is also a ray-tracer, but adopts a Monte Carlo photon-packet technique. Finally, RSPH is the only code from IL06 to operate directly on SPH fields; its results are nonetheless provided as a uniform grid.

[4] See `http://www.cita.utoronto.ca/~iliev/rtwiki/doku.php`, accessed 2017-06-27, and `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project/index.html`, accessed 2017-07-18 (previously `https://astronomy.sussex.ac.uk/~cs390/RT_comparison_project/index.html`, accessed 2016-04-21)

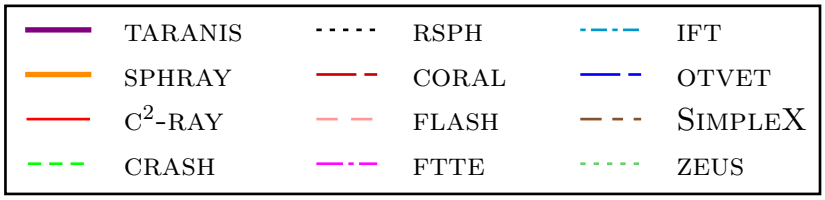


Figure 5.1. CRTCP Tests results legend.

5.1 Test I: pure hydrogen isothermal HII region expansion

Test I is the only test for which an analytic solution is available. It models the growth of a Strömgren sphere in a uniform density, isothermal, pure hydrogen background over 500 Myr. The number density is $n_{\mathrm{H}} = 1.0 \times 10^{-3}\,\mathrm{cm}^{-3}$, with an initial ionization fraction $x_{\mathrm{HII}} = 1.2 \times 10^{-3}$ and temperature $T = 10^4\,\mathrm{K}$. A monochromatic source emits $\dot{N}_\gamma = 5 \times 10^{48}$ photons per second, all with energies equal to the ionization threshold of hydrogen. In IL06, this source is placed at the corner of a box of side-length 6.6 kpc. To avoid edge-effects due to a lack of period boundary conditions in the ray tracing scheme,[5] here the source is instead placed at the centre of a 13.2 kpc box. Without this modification, erroneously-low optical depths are computed near the volume bounds, and hence ionization fronts in those regions move faster than expected.

The test is specified as isothermal, with a constant temperature $T = 10^4\,\mathrm{K}$. To that end, entropy is reset accordingly at the end of each timestep. (Note that entropy is not itself constant, being a function of both temperature and the number of free particles.)

Initial conditions

A 128^3 gas-particle glass-like dataset is used as the initial condition. This was provided by Gabriel Altay (SPHRAY author) and made publicly available.[6] It has been produced using the GADGET-2 code, the glass-generating method of which requires periodic boundary conditions, hence the need to avoid edge-effects. IL06 specify a 128^3 cell grid for their test. Here, a 128^3 particle count is assumed to be consistent, but note that particle counts of two- to three-times larger may be required to achieve equivalent resolution in hydrodynamical problems (Hubber et al., 2013). Further, note that doubling

[5] GRACE simply does not yet implement support for periodic boundary conditions.

[6] `https://code.google.com/archive/p/rt-comparison-project-snapshots/`, accessed 2017-02-05.

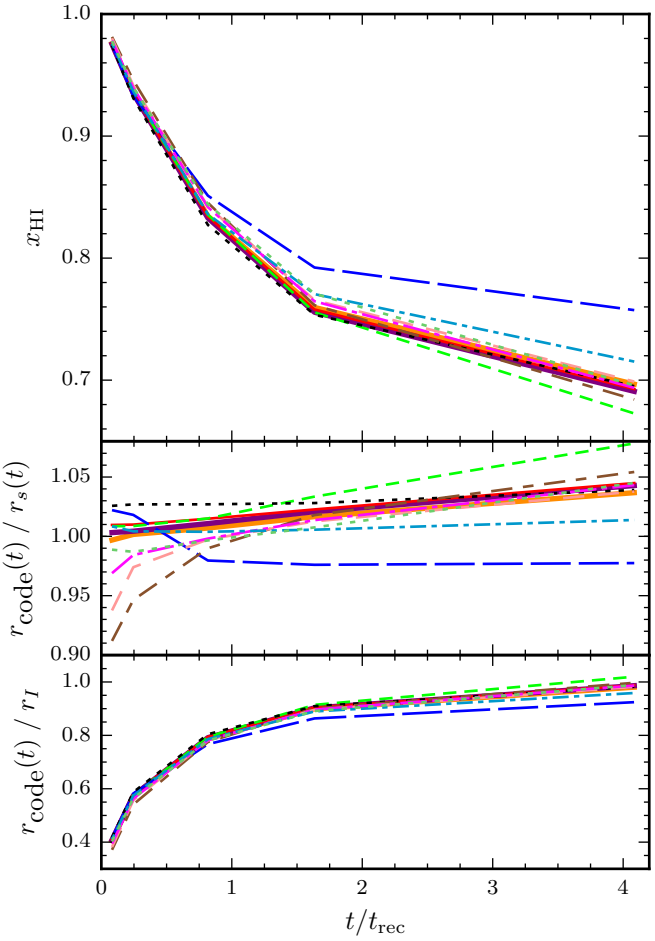


Figure 5.2. *Top:* Test I mean volume-weighted neutral fraction over time. *Middle:* The front radius, where $x_{\mathrm{HII}} = 0.5$, reported as a fractional of the analytic result at that time. *Bottom:* The same front radius, reported as a fraction of the numerically-integrated equilibrium radius.

the side-length of the volume effectively halves the resolution in the radial (distance from source) direction; tests with a central source thus have equivalent spatial radial resolution to a 64^3 particle volume with a corner source.

Results and discussion

Here, TARANIS traces 4096 rays per timestep, a number which is more than sufficient for convergence.

The mean volume-weighted neutral-fraction and ionization-front radius over time are shown in Figure 5.2. The front radius is defined to be the point at which $x_{\mathrm{HII}} = 0.5$. The latter value is expressed in terms of both the analytical front radius radius, $r_s(t)$, and the numerically-integrated equilibrium radius $r_I \equiv r(x_{\mathrm{HII}} = 0.5,\, t \to \infty)$.

For r_s, we have (IL06)

$$r_s(t) = R\left[1 - \exp\left(\frac{-t}{t_{\rm rec}}\right)\right]^{1/3},$$

where R is the Strömgren radius,

$$R \equiv \frac{3\dot{N}_\gamma}{4\pi\,\alpha^B_{\rm HII}(T)\,n^2_{\rm H}}$$

and $t_{\rm rec}$ is the recombination time,

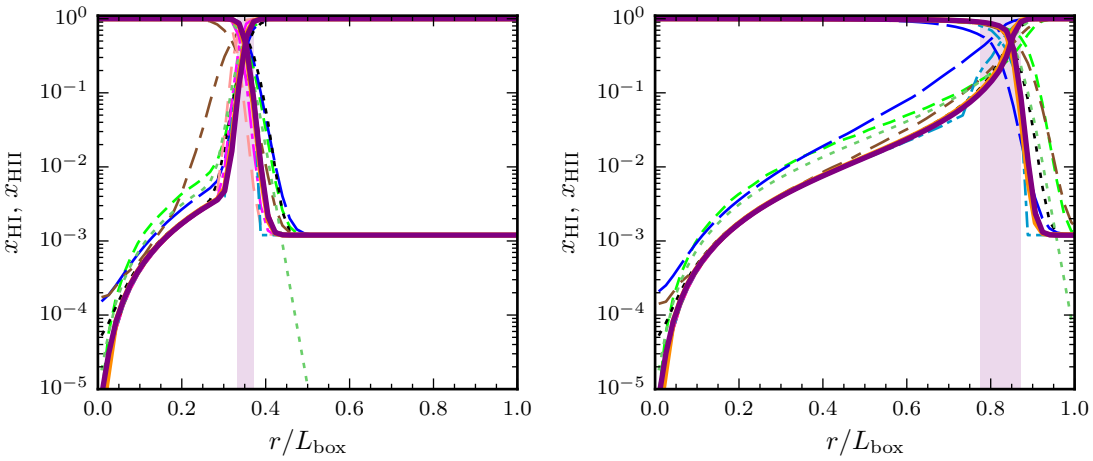
$$t_{\rm rec} \equiv \frac{1}{\alpha^B_{\rm HII}(T)\,n_{\rm H}}.$$

The values of IL06 are assumed, $\alpha^B_{\rm HII}(10^4\,{\rm K}) = 2.59 \times 10^{-13}\,{\rm cm^3 s^{-1}}$, thus $t_{\rm rec} = 122.4\,{\rm Myr}$ (c.f. $\alpha^B_{\rm HII}(10^4\,{\rm K}) = 2.548 \times 10^{-13}\,{\rm cm^3 s^{-1}}$ from Table 5.1, thus $t_{\rm rec} = 124.5\,{\rm Myr}$).

The equilibrium front radius is computed using the RABACUS solver (Altay and Wise, 2015), with the physical specifications of test I assumed over 512 layers. (A helium number density is required, which is set to $1 \times 10^{-15}\,{\rm cm^{-3}}$.)

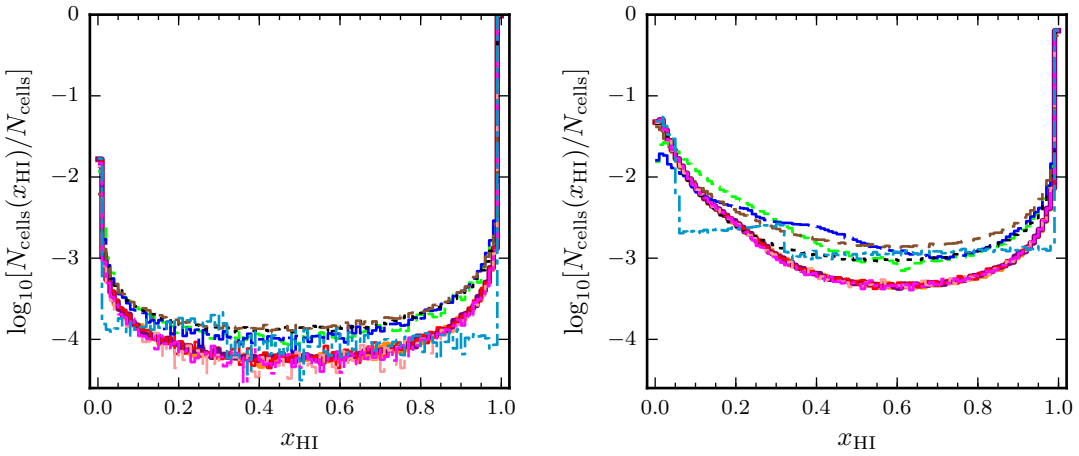
Excellent agreement between SPHRAY and TARANIS is seen in all cases. The ionization front radius is never more than 5% from the analytic value, and more importantly, this error is less than the width of the ionization front at late times (see Figure 5.3). Comparing to the equilibrium radius, we see that TARANIS matches this extremely well — in fact, $r_I \simeq 1.05R$ (e.g. Pawlik and Schaye, 2008), hence why most codes overestimate r_s at late times. TARANIS also agrees well with the other ray-traced codes, excepting CRASH, which appears to suffer from substantial shot-noise and is not as well converged.

Spherically-averaged, mass-weighted radial profiles are shown in Figure 5.3. Again, TARANIS follows very well the profiles of SPHRAY and other comparable codes, and in particular IFT, whose numerical method is based on the analytic result of exactly the Test I problem (but which is not accurate where non-equilibrium chemistry is significant, i.e. at the ionization front). This holds also for intermediate times not shown in Figure 5.3. IL06 note an analytic front width of ~ 14 grid cells, or approximately $0.11L_{\rm box}$, which is reproduced by TARANIS at late times. Again CRASH differs here, but AL08 showed that a wider ionization front is indicative of poor convergence (in fact, their non-converged profiles match very well those of CRASH). The other codes are more diffusive in nature and TARANIS has an expectedly sharper ionization front.



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 500\,\mathrm{Myr}$.

Figure 5.3. Test I radially-averaged ionization fraction profiles. $L_{\rm box} = 6.6\,\mathrm{kpc}$ as in IL06. The shaded area denotes the region $0.1 \leq x_{\rm HI} \leq 0.9$ for TARANIS.



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 500\,\mathrm{Myr}$.

Figure 5.4. Test I, histograms of the fraction of cells with a given neutral-fraction.

Finally, in Figure 5.4 histograms of cells (particles) with a given neutral fraction are shown. ZEUS results are omitted, because cell values are not well distributed and the resulting histograms obfuscate those of the other codes. Again, TARANIS, SPHRAY and most other ray tracing codes are in excellent agreement. Note, however, that RSPH appears more in-line with the more diffuse implementations, an undesirably property avoided by both TARANIS and SPHRAY. This is also visible in the post-front profiles of Figure 5.3.

5.2 Test II: HII region expansion: the temperature state

Test II is essentially a physically-accurate variant of test I. The gas temperature is now allowed to vary, and the source is modelled as a $T = 10^5\,\mathrm{K}$ black body, with the same hydrogen-ionizing photon emission rate $\dot{N}_\gamma = 5 \times 10^{48}\,\mathrm{s}^{-1}$. RT parameterizes its black body luminosity function instead in terms of effective temperature and surface area, here $4.68 \times 10^{18}\,\mathrm{m}^2$; the conversion is detailed in Appendix A. The test begins with a hydrogen-only initial condition of equal density to that of test I, but it is fully neutral and has a temperature of $T = 100\,\mathrm{K}$. (In TARANIS, we actually begin with a neutral fraction of $x_{\mathrm{HI}} = 1 - 10^{-5}$ to avoid extremely-short minimum timesteps when ionizing 'unionized' particles.) Again, for TARANIS and SPHRAY the source is placed at the centre, rather than corner, of the box.

The input particle distribution is identical to that used for test I.

The version of SimpleX used in IL06 does not track the temperature state, and is not included in this test. The version of ZEUS used in IL06 supports only a monochromatic spectrum for the source. No FLASH data has been made available for this test.

Results and discussion

Again, TARANIS traces 4096 rays per timestep and is well converged.

The mean volume-weighted neutral fraction over time is shown in Figure 5.5. The introduction of variable temperatures results in more variation between codes, but TARANIS still closely tracks the results of SPHRAY and C^2-RAY. RSPH and CRASH are also similar here. Additionally shown in Figure 5.5 is the ionization front as a fraction of the equilibrium ionization front radius, as computed by the RABACUS solver. For the latter, the physical specifications of test II are assumed over 512 layers. Photons with frequencies in $[\nu_o, 50\,\nu_0]$ are integrated over; equations in Appendix A show that this covers a fraction $> 1 - 3 \times 10^{-5}$ of the total ionizing photon count of the black body. At late times, TARANIS and the other ray tracing codes have ionization fronts within a few percent of the semi-analytic solution.

The temperature structure at early times is shown in Figure 5.6. For the CRTCP codes, slices of one cell-width are shown, taken from a simulation box face which includes the source corner. For both TARANIS and SPHRAY, the datasets are first interpolated onto a 256^3 cell grid according to the SPH weights; a 128^3 octant is then selected and

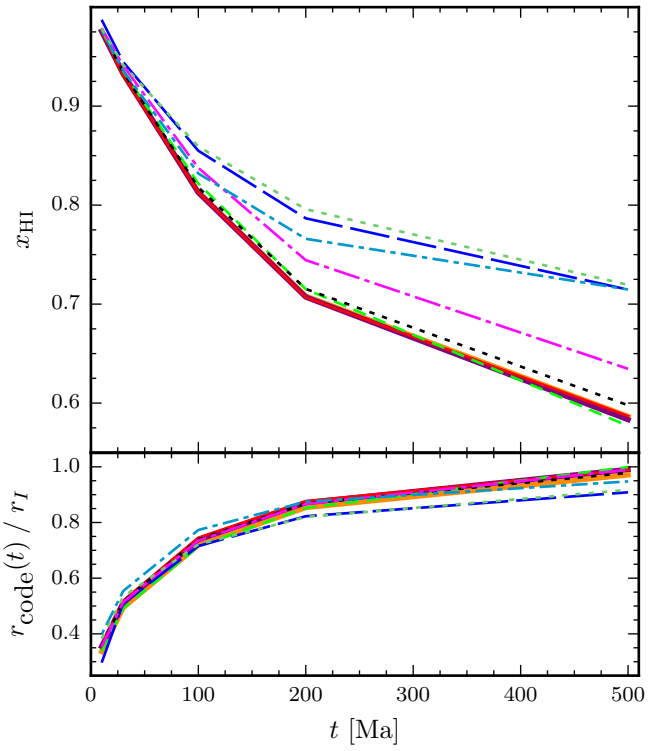


Figure 5.5. Test II mean volume-weighted x_{HI} and ionization front radius over time. The front radius is the point at which $x_{\mathrm{HI}} = 0.5$, reported as a fractional of the RABACUS equilibrium result.

treated identically to the CRTCP grids. Note that, again, the linear resolution of the SPH results is effectively halved. The OTVET, FTTE, IFT and ZEUS results show no heating ahead of the ionization front, by design (IFT) or due to a lack of high-energy photons. Note that ZEUS here is monochromatic, and thus highlights the effects of ignoring the spectral nature of the sources. Pre-front heating in TARANIS closely matches that of RSPH, though CRASH, SPHRAY and, to a slightly lesser extent, C^2-RAY are all similar.

Shot noise is present in the TARANIS results, but at a significantly-reduced level compared to SPHRAY and particularly CRASH, which again is not well-converged. (IL06 note that the multi-frequency treatment of photon packets employed by CRASH has been modified 'for production runs', presumably resulting in better convergence than shown here and throughout.)

These results generally hold at intermediate times (not pictured), where TARANIS most closely mirrors RSPH and is generally similar to those ray traced codes which model high-energy photons. At 500 Myr (not pictured), all ray-traced codes show a near-uniform temperature map of $T \simeq 10^4$ K; C^2-RAY still maintains the hottest source-proximity region, and a cooler patch at the corner farthest from the source, while RSPH is the most uniform, and SPHRAY and TARANIS lie between.

Spherically-averaged, mass-weighted radial profiles are shown in Figure 5.7 (neutral and ionized fractions) and Figure 5.8 (temperature). The ionization state profiles of TARANIS are in good agreement with those of the other ray traced codes, particularly at late times where TARANIS, RSPH and C^2-RAY results are nearly identical. SPHRAY results here have a consistently wider ionization front, with the $x_{\mathrm{HII}} = 0.5$ point slightly closer toward the source.

It should be noted that AL08 present marginally different results: in the ionized region behind the front, their SPHRAY profiles lie between OTVET and RSPH; in the unionized region beyond the front, their profiles most closely follow C^2-RAY. With the exception of the region $r/L_{\mathrm{box}} \gtrsim 0.5$ at $t = 10$ Myr, the profiles of AL08 more closely match TARANIS than those reproduced here. This is likely due to changes made to the SPHRAY codebase post-publication, in particular (unspecified) improvements to its sampling of high-energy photon packets (G. Altay, private communication, 2012-11-30). This is consistent with results seen in Section 5.1 for a monochromatic source, which are identical to AL08, and with the heating seen in Figure 5.8 (discussed below). Changes

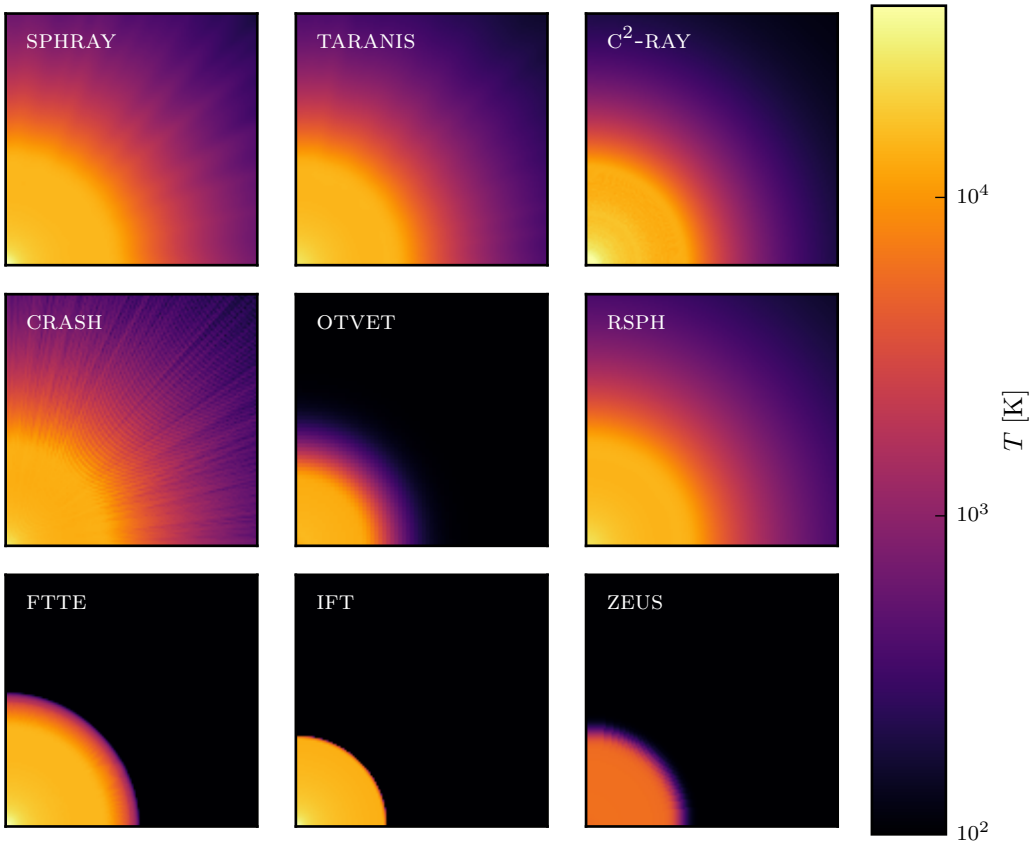
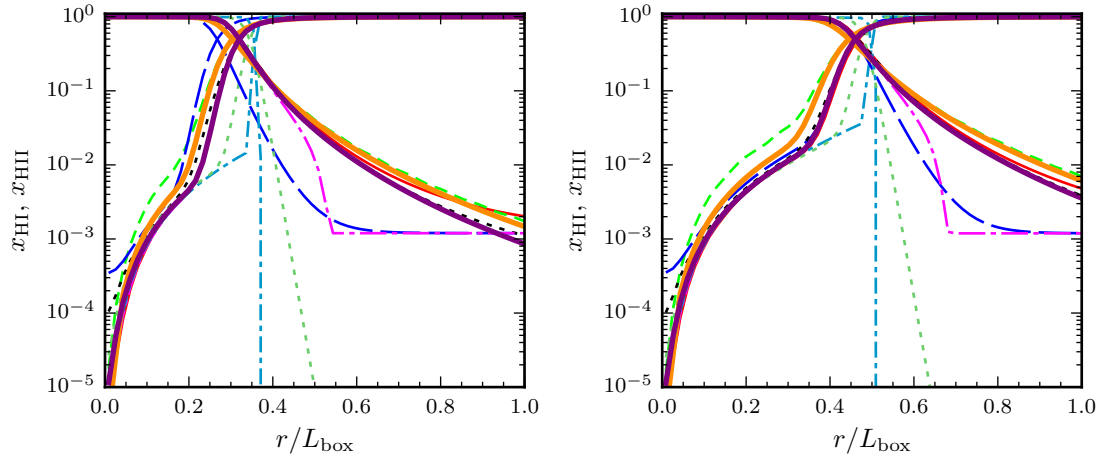
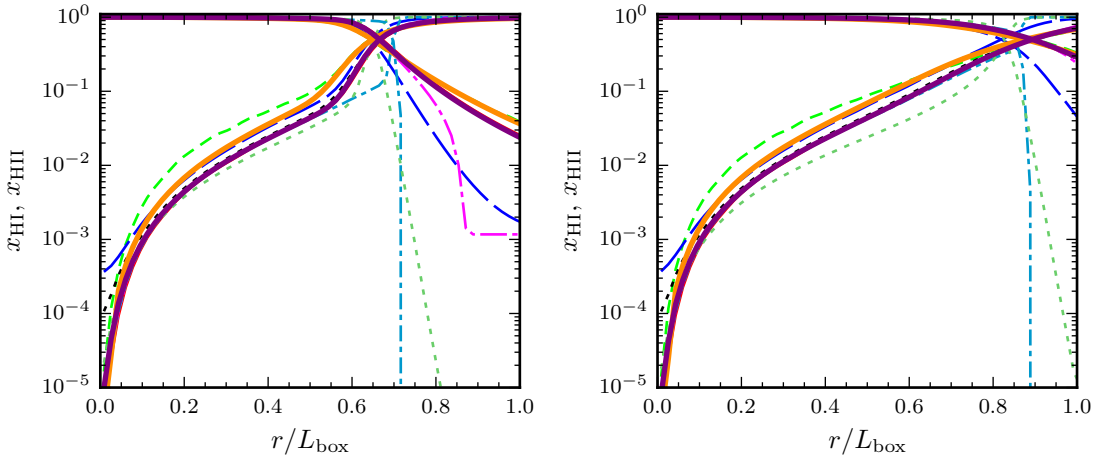


Figure 5.6. Test II temperature slices at $t = 10\,\mathrm{Myr}$. Slices are one cell in width, or $0.052\,\mathrm{kpc}$ in all cases.



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 30\,\mathrm{Myr}$.

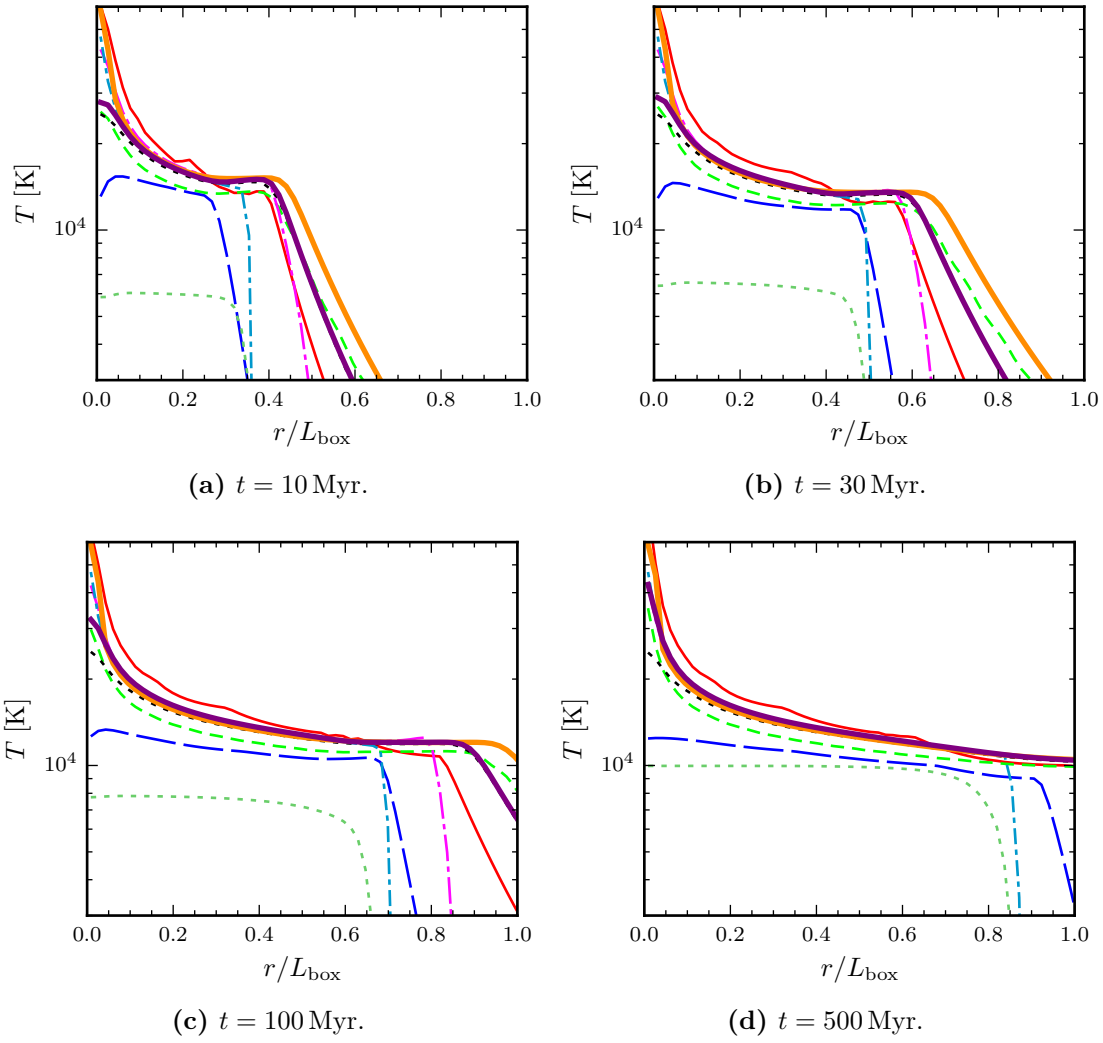


(c) $t = 100\,\mathrm{Myr}$. **(d)** $t = 500\,\mathrm{Myr}$.

Figure 5.7. Test II radially-averaged ionization fraction profiles. $L_{\mathrm{box}} = 6.6\,\mathrm{kpc}$ as in IL06.

have also been made to the numerical solver, which is now (by default) a backward Euler scheme. Nonetheless, these explanations remain somewhat unsatisfactory and not readily verified. TARANIS, C^2-RAY and RSPH all integrate over the full frequency range; AL08 state that, as published, SPHRAY has an "accurate treatment of high-energy photons"; and the numerical solvers for SPHRAY as published are no longer available without reimplementation.

A similar situation is to be found in the temperature profiles. All ray-traced codes behave similarly, but the differences are more pronounced than previously. TARANIS again follows RSPH extremely closely, except in close proximity to the source, where it has increased temperatures by a factor of a few; all codes show variation here, but converge into a much narrower temperature band within $\sim 0.1 r/L_{\mathrm{box}}$. At intermediate times we



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 30\,\mathrm{Myr}$.

(c) $t = 100\,\mathrm{Myr}$. **(d)** $t = 500\,\mathrm{Myr}$.

Figure 5.8. Test II radially-averaged temperature profiles. $L_{\mathrm{box}} = 6.6\,\mathrm{kpc}$ as in IL06.

see heating extending farther beyond the front in SPHRAY, while AL08 produced profiles closely following those of RSPH, and hence TARANIS. Potential causes for this change are as discussed above. In the ionized region behind the front, TARANIS and SPHRAY compute very similar equilibrium temperatures.

Since the radial profiles at $t = 500\,\mathrm{Myr}$ should be (very close to) the equilibrium state, we may also compare to the equilibrium ionization and temperature profiles of RABACUS (obtained as for the ionization front position). These are shown in Figure 5.9. In the ionized region, TARANIS shows excellent agreement with the RABACUS ionization results, while beyond the front RABACUS predicts slightly more ionization than either of the codes. Temperatures are mostly consistent among all three, but with RABACUS also producing the cooler temperatures of TARANIS in proximity to the source. At distance

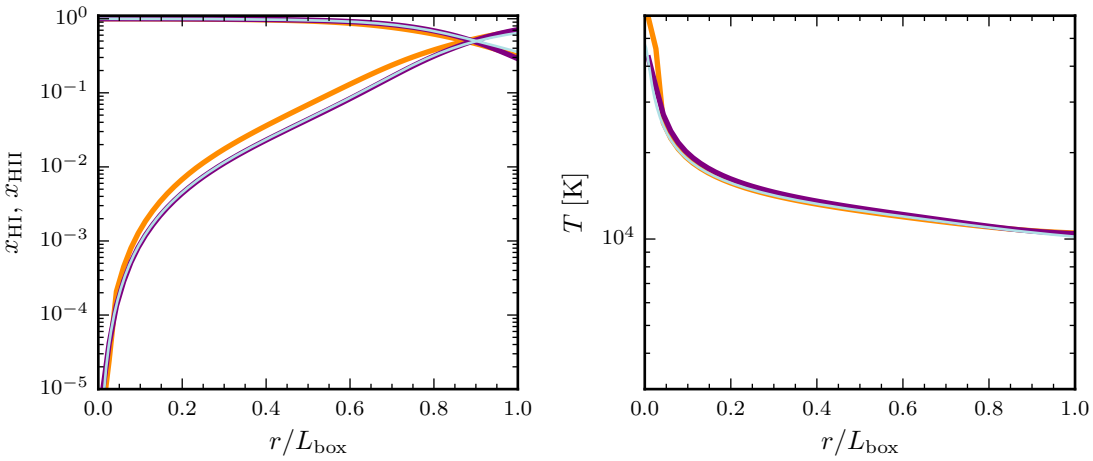


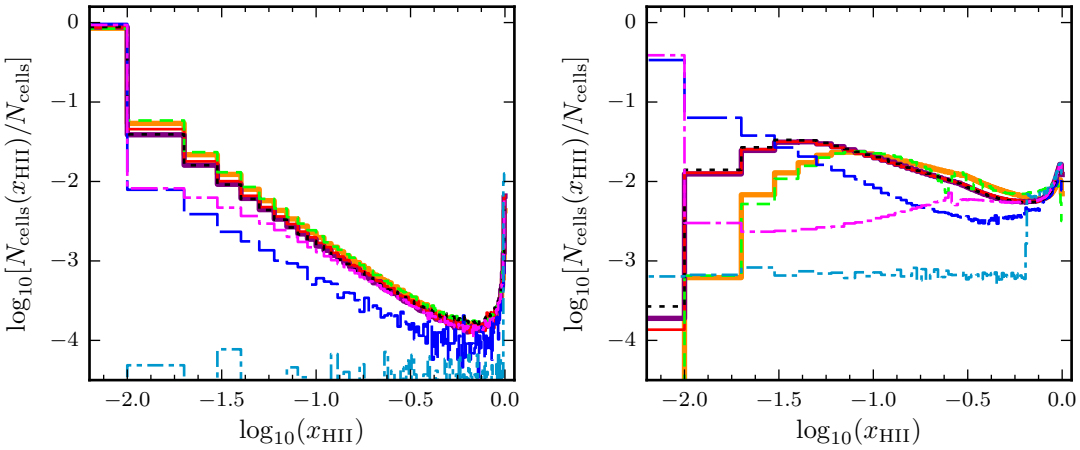
Figure 5.9. Test II radially-averaged $t = 500\,\mathrm{Myr}$ ionization fraction and temperature profiles for TARANIS and SPHRAY, and equilibrium profiles from RABACUS. RABACUS is in light-blue.

from the source, SPHRAY and RABACUS are marginally cooler; this small difference may be attributable to the different rates adopted, which are largely identical between those two packages.

Finally, histograms of ionization state and temperature are given in Figure 5.10 and Figure 5.11, respectively. For reasons noted in the test I results, ZEUS histograms are again omitted. IFT ionization histograms here differ from those of IL06, a point discussed in Appendix E.

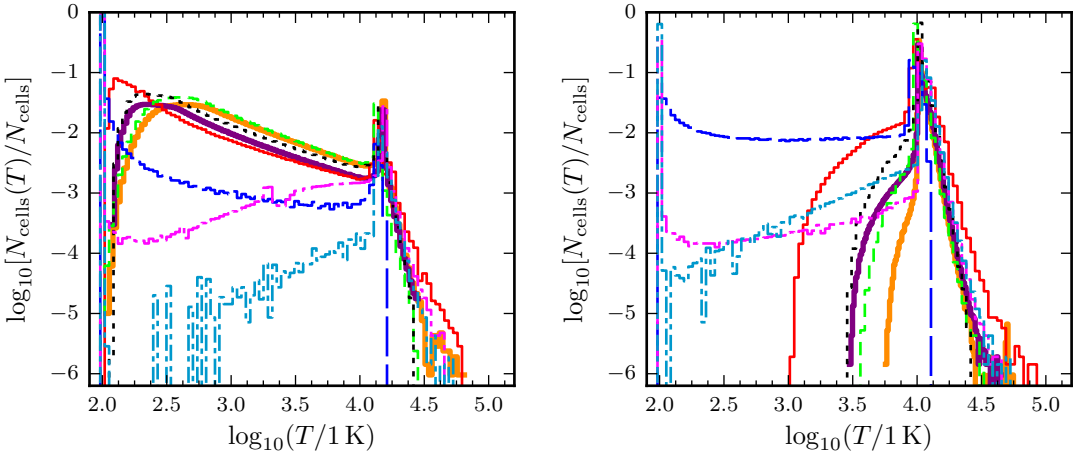
At early times, where the ionization front is fast-moving, all ray-traced codes are consistent, a trend which holds through intermediate times ($t \simeq 200\,\mathrm{Myr}$, not shown). At late times, the Monte Carlo photon packet codes SPHRAY and CRASH converge to a similar result, and the remaining ray tracers converge to another. The former see a slightly reduced fraction of almost-neutral gas, with a corresponding increase in the $-0.8 \lesssim \log_{10}(x_{\rm HII}) \lesssim -0.2$ range. That is, the photon packet codes have more gas in an intermediate ionization state, consistent with the wider fronts seen in the profiles.

Temperatures are less consistent, even at early times. TARANIS agrees well with all codes at the peak temperature, which is a little above $10^4\,\mathrm{K}$ at early times and at $10^4\,\mathrm{K}$ at late times. The highest temperatures, found near the source, are similar between TARANIS and RSPH, as was seen in the radial profiles. Unlike SPHRAY, TARANIS does not achieve temperatures $T \simeq 10^{4.5}\,\mathrm{K}$ until $t \simeq 200\,\mathrm{Myr}$ (not shown), and so we again see a greater impact from the high-energy photon sampling of SPHRAY. Note that, as already mentioned, increasing to 10^9 rays for SPHRAY does not alter this feature,



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 500\,\mathrm{Myr}$.

Figure 5.10. Test II, histograms of the fraction of cells with a given ionized fraction.



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 500\,\mathrm{Myr}$.

Figure 5.11. Test II, histograms of the fraction of cells with a given temperature.

so it is not indicative of a lack of convergence. Convergence is also demonstrated for TARANIS in Section 5.5, though this does not rule out inaccuracy in the integration scheme used to compute photoionization and photoheating rates. A significant fraction of $T \lesssim 10^4\,\mathrm{K}$ gas is visible in C^2-RAY, largely lying at radii beyond L_{box} and hence not previously shown; this suggests that the C^2-RAY scheme may not be correctly heating gas distant from the source, and that TARANIS and the other ray tracers lie closer to the true solution. Different cooling rates also likely account for some of these differences, and in any case, the total fraction of cells covered in the lower-temperature region is small.

5.3 Test III: Ionization front trapping in a dense clump and the formation of a shadow

In test III a cold, dense spherical clump of neutral hydrogen is embedded in a hot, uniform, and much less dense neutral hydrogen background. The box side-length is again 6.6 kpc, which here is unmodified as edge-effects are of no concern. The clump is placed at $\vec{x} = (5,\, 3.3,\, 3.3)$ kpc and of radius $r = 0.8$ kpc. Outside the clump, the number density of hydrogen is $n = 2 \times 10^{-4}\,\mathrm{cm}^{-3}$, and the temperature is $T = 8 \times 10^3\,\mathrm{K}$. Inside the clump, $n_{\mathrm{clump}} = 200n$, and $T_{\mathrm{clump}} = T\,/\,200$. The $x = 0$ face of the simulation box is taken to be a black body source, with effective temperature $T_{\mathrm{eff}} = 10^5\,\mathrm{K}$ and ionizing photon-count flux $F = 10^6\,\mathrm{s}^{-1}\mathrm{m}^2$. The test evolves this system for 15 Myr.

IL06 have chosen these parameters such that the ionization front is expected to become trapped at approximately the mid-point of the clump.

SimpleX did not complete this test.

Initial conditions

Creating an acceptable input particle distribution for this test proved somewhat difficult.

Initially, a 128^3 glass-like field in a 6.6 kpc box was modified as follows. First, note that an overdensity factor $f = 200$ is easily produced by increasing the SPH-particle number density in that region by the same factor. Assuming for the moment that we have a glass of sufficently-large extent to sample from, we may then select some spherical region of exactly the size to contain $f - 1$ more particles than are currently present at the location of the clump. All particles within this spherical region are copied, and scaled such that we have a new sphere, of equal radius to the clump, and overdensity $f - 1$. The particles are then simply overlaid in the clump position to achieve the required SPH-particle number density.

Three practical issues then arise. First, the starting glass is not large enough to contain a sphere of the required size; second, the final number of particles well exceeds the original 128^3; and third, the smoothing lengths within the clump are not consistent (1 in every 200 particles has a smoothing length equal to that of the low-density background). Finally, note that cutting or overlaying glass SPH fields will destroy the glass-like properties at the cut boundaries and in the overlap regions. This makes all distributions

generated here unsuitable for hydrodynamical tests. For a method appropriate to dynamical tests, see for example Agertz et al. (2007).

The first issue is solved by taking several smaller-radius spheres, for example, four spheres containing only $\sim f\,/\,4$ more particles than initially exist at the clump location. The second issue is solved by using, as the starting glass, only a sub-region of the original 128^3 glass. This region is chosen such that it contains only $\sim 8.44 \times 10^5$ particles, rescaled to retain a box side-length of 6.6 kpc. The clump then requires an additional $\sim 1.25 \times 10^6$ particles be added, resulting in a total particle count of $2.09 \times 10^6 \sim 128^3$.

The resulting file is then passed as a particle position-only initial condition for the GADGET-2 code, which is able to compute smoothing lengths given a target number of near-neighbours; $N_{\mathrm{ngb}} = 32$ is set here. The first GADGET-2 output, before any hydrodynamic timesteps have been taken, is the final $\sim 128^3$ particle distribution.

This procedure is moderately successful, but smoothing radii in the clump show a bi-modal distribution, and the number of near neighbours there is typically lower than the target. To counter this, in the final stage the minimum-allowed smoothing radius (`MinGasHsmlFraction`) is increased from 0 to 0.1 (found by trial-and-error). The resulting smoothing radii, within the clump, are then found to be all equal, having reached the minimum, at 0.028 kpc. Near-neighbour counts now vary from $\sim 30 - 60$, indicative of noise in the clump's particle distribution. (Outside the clump, smoothing radii are unchanged and as expected in all cases.) This latter dataset is used, on the basis that it does not contain too-low near-neighbour counts. In any case, no 'ideal' distribution, with consistent neighbour counts in the clump, was found to be produced, and when arbitrarily-small smoothing radii were allowed, GADGET-2 failed to converge when computing them. This initial condition is codified as test3-128, containing 2 096 871 particles.

In light of the above, a second particle distribution was generated as follows. Beginning with the same 128^3 particle glass, in a 6.6 kpc side-length box, we first remove all particles in the region of the clump. Next, a single spherical region (whose size is again such that the required particle count is achieved) is selected from a 256^3 glass dataset, scaled to the size of the clump, and placed there. This results in a larger, $\sim 170^3$ particle dataset, but the glass-like properties will be preserved within the clump. Smoothing radii are then recomputed as follows. First, all radii are assumed smaller than that

expected in the clump; we then iterate, looping over all particles and increasing their radii by a small fraction (~ 1.2) if the number of near-neighbours is less than the target, $N_{\rm ngb} = 32$. This initial condition is codified as test3-170-smooth, containing 5 210 233 particles.

For the final dataset, we follow the same basic procedure, but take a cut of the starting 128^3 dataset, and a smaller number of particles from the 256^3 glass, such that the final number of particles is approximately 128^3. This dataset is codified as test3-128-smooth, containing 2 096 801 particles.

Note that, in all cases, higher spatial resolution is attained within the clump than expected for a uniform-grid as was used in the CRTCP. This is of course an intended consequence of the SPH method, but should be kept in mind for the following analysis.

Results and discussion

Here, TARANIS traces 1024 rays per timestep, and a timestep multiplier of $f = 0.5$ (recall Section 4.5) which is sufficient for convergence. All results shown here, for both TARANIS and SPHRAY, correspond to the test3-128-smooth particle field. Differences observed from runs using the test3-128 and test3-170-smooth fields are detailed at the end of this section.

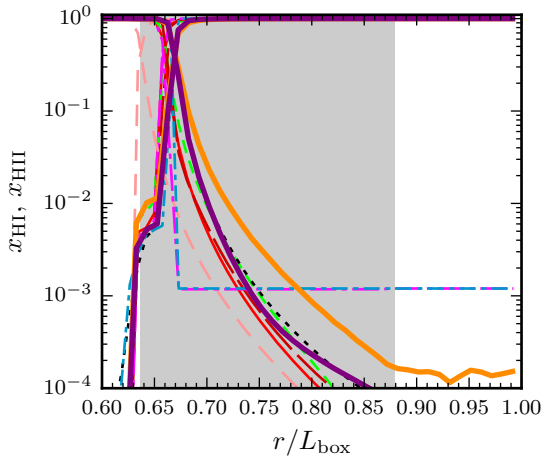
We begin with ionization and temperature profiles, shown in Figure 5.12. For the CRTCP codes, a $128 \times 2 \times 2$ cell column, centred in the y-z plane, is averaged over. For TARANIS and SPHRAY, all particles whose centres lie within an equivalent column of side-length 0.1 kpc are averaged over. Note that AL08 instead choose all particles within a 0.05 kpc cylinder; this alternative was found to have no noticeable bearing on the resulting profiles. Interpolating the SPHRAY and TARANIS data onto a 128^3 grid and averaging over the $128 \times 2 \times 2$ central column also gives comparable profiles, albeit substantially noisier. At all times, TARANIS agrees with the majority of codes in the ionized region behind the front. SPHRAY predicts a higher neutral fraction within the clump, but also a higher ionized fraction beyond it, though TARANIS, SPHRAY and RSPH remain in agreement on the location of the point $x_{\rm HII} = 0.5$. (Interestingly, at late times, the SPHRAY results of AL08 found this crossing point to occur approximately $0.2L_{\rm box}$ closer to the source. They also identified a neutral fraction beyond the clump slightly in excess of $x_{\rm HI} = 10^{-2}$, similarly to TARANIS.) TARANIS also preserves a sharp transition in the neutral fraction at the source-facing clump boundary, while RSPH appears to

show some diffusion. Both C^2-RAY and particularly CRASH predict front-trapping closer to the middle of the clump than TARANIS; examining the corresponding temperatures, we see that this is somewhat consistent with their colder clump interiors. As has been seen previously, the temperature state appears to show more variation between codes; TARANIS is most consistent with RSPH here, in particular regarding the depth to which equilibrium temperatures penetrate the clump. Finally, at late times, TARANIS begins to heat the shielded region above its initial temperature, an indication that the clump is optically thin (or at least thinner) to photons in the high-energy tail.

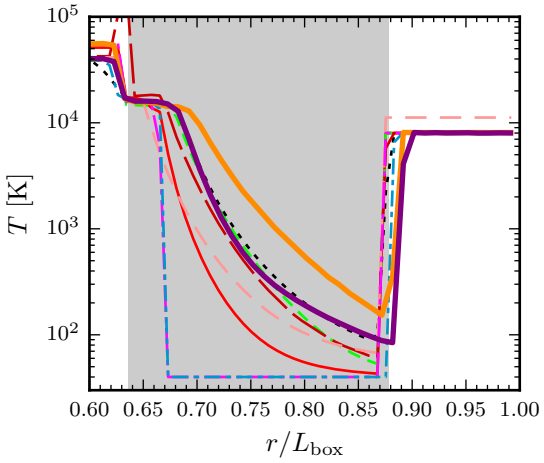
Much as for test II, it is also reasonable to compare the late-time profiles to those produced by RABACUS. Here, the slab model is used, consisting of 512 layers with properties equal to that of the clump, and the same incident ionizing flux is assumed, integrated over $[\nu_0, 50\nu_0]$. Note that RABACUS does not support non-uniform media, hence only the state within the clump is shown in Figure 5.13. In the ionized region behind the front, TARANIS and RABACUS are in good agreement, as TARANIS was with several of the other codes. Beyond the front, however, RABACUS computes almost an order of magnitude greater ionized fraction. It is plausible that, at $t = 15\,\mathrm{Myr}$, neither TARANIS nor SPHRAY have quite reached ionization equilibrium; this has not been further investigated. Temperatures within the clump are all in excellent agreement, though the small, colder patch retained by TARANIS would again be consistent with it not having yet reached its equilibrium state in this dense medium.

Moving on to the wider clump ionization-state structure, a one-cell width slice of the neutral fraction is shown in Figure 5.14. To produce these figures for SPHRAY and TARANIS, the SPH results were first interpolated onto a 128^3 grid. Note first that all ray-traced codes appear to show some noise within the clump. The SPH codes additionally show, unsurprisingly, significantly more diffusion around the shadow edges; this was also true at early times (not pictured), where the grid codes still preserved a sharp shadow. Nonetheless, TARANIS still preserves a distinct shadow, similarly to SPHRAY. This test identifies a weakness of RSPH, with its more diffusive method creating an anomalously-large shadow transition.

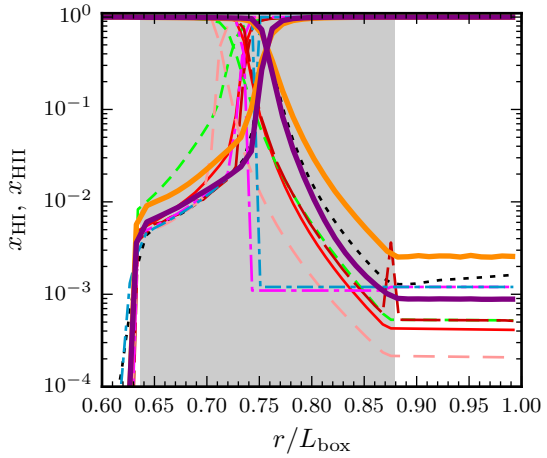
The corresponding temperature slice is given in Figure 5.15. At early times (not shown), substantial shot noise is seen within the clump for SPHRAY, while TARANIS has a particularly smooth transition. And again, the SPH codes show diffusion at the shadow edges not present for the grid codes. At late times, some differences in the



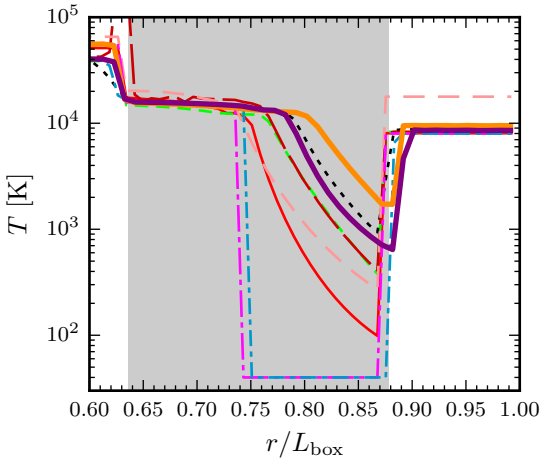
(a) Ionization state at $t = 1\,\mathrm{Myr}$.



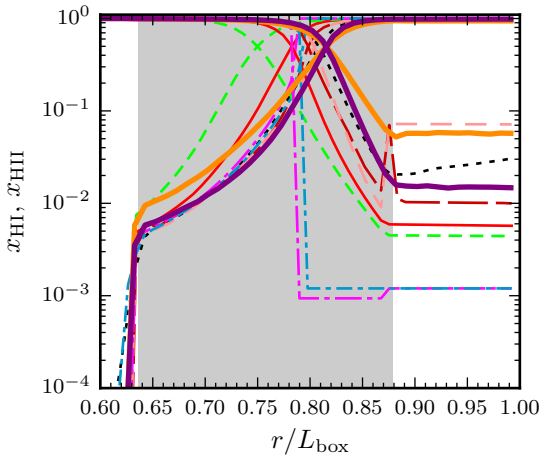
(b) Temperature state at $t = 1\,\mathrm{Myr}$.



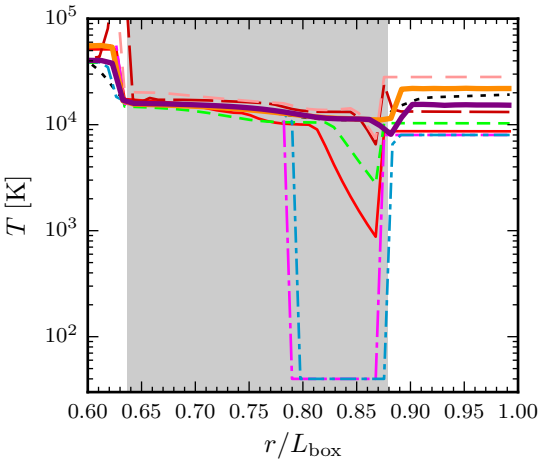
(c) Ionization state at $t = 5\,\mathrm{Myr}$.



(d) Temperature state at $t = 5\,\mathrm{Myr}$.



(e) Ionization state at $t = 15\,\mathrm{Myr}$.



(f) Temperature state at $t = 15\,\mathrm{Myr}$.

Figure 5.12. Test III averaged ionization and temperature profiles. $L_{\mathrm{box}} = 6.6\,\mathrm{kpc}$ as in IL06. The vertical band denotes the extent of the clump.

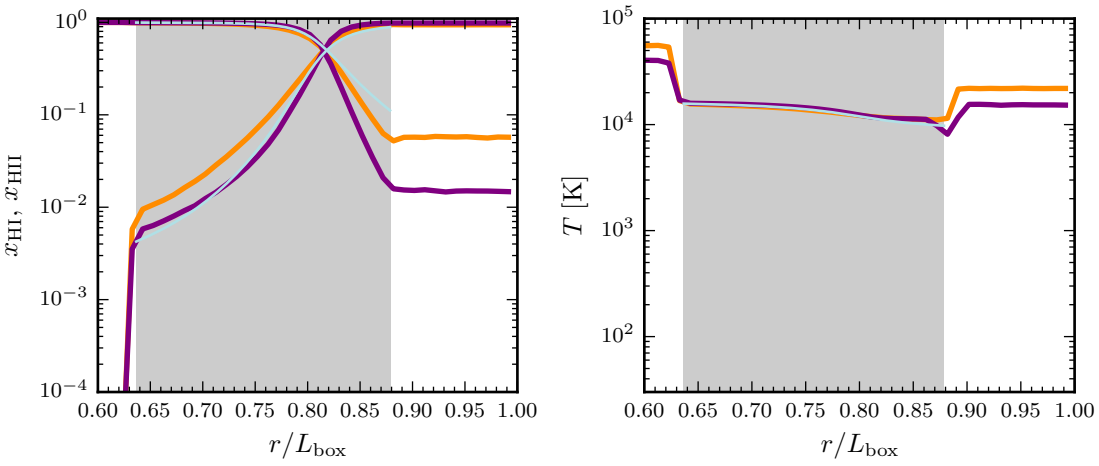


Figure 5.13. Test III averaged $t = 15\,\mathrm{Myr}$ ionization fraction and temperature profiles for TARANIS and SPHRAY, and equilibrium profiles from RABACUS. RABACUS is in light-blue.

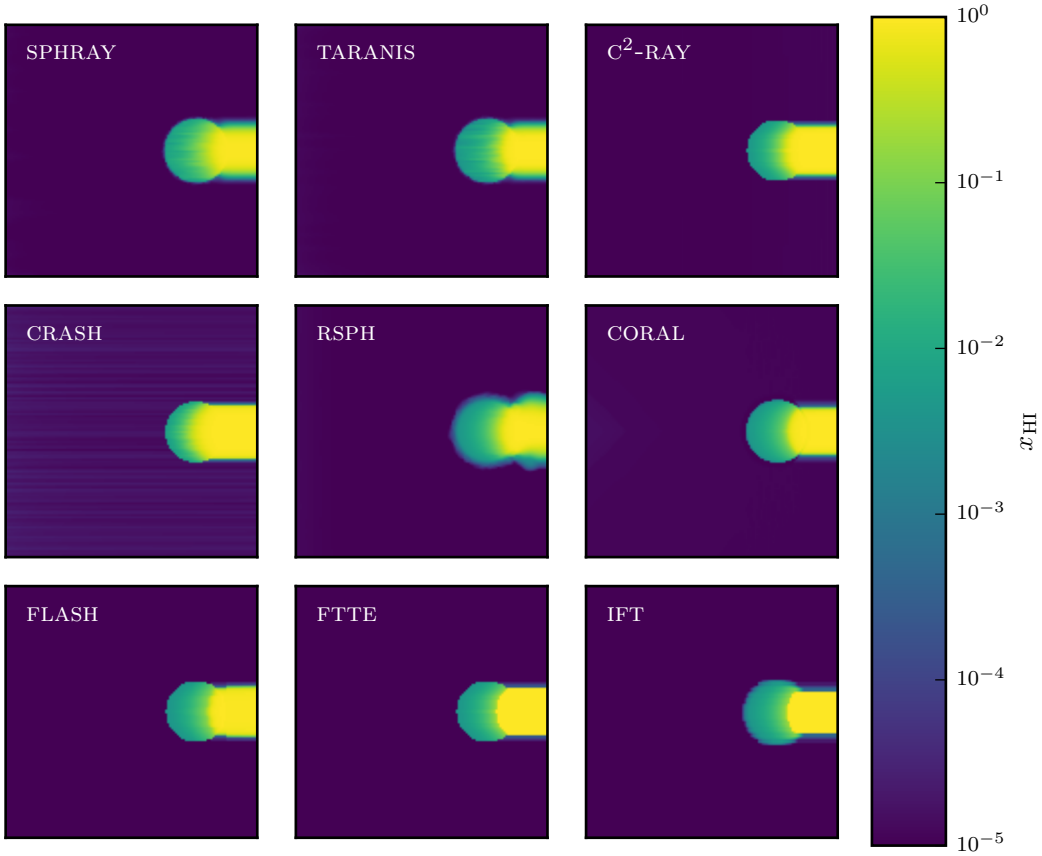


Figure 5.14. Test III neutral-fraction slices at $t = 15\,\mathrm{Myr}$. Slices are one cell in depth, or 0.052 kpc in all cases.

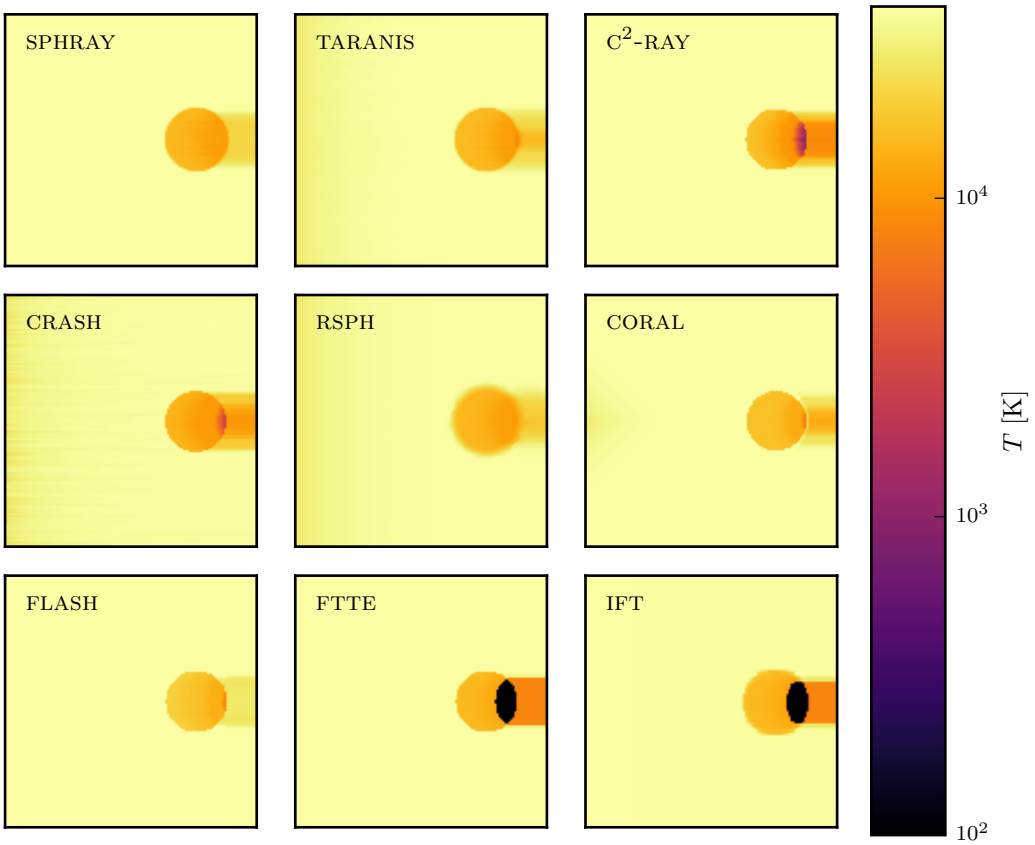


Figure 5.15. Test III temperature slices at $t = 15\,\mathrm{Myr}$. Slices are one cell in depth, or 0.052 kpc in all cases.

shadow structure become apparent. Banding in the shadow's temperature is visible for TARANIS, consistent with those codes which also retain a colder patch at the right-most side of the clump. Once again, TARANIS here is in fact very similar to the SPHRAY results published by AL08.

TARANIS maintains a sharp clump boundary, as does SPHRAY, but RSPH is again shown to be substantially more diffusive.

Histograms of the ionization state inside the clump, at late times, are shown in Figure 5.16. All particles and cells whose centres lie within 0.8 kpc of the centre of the clump are included. Both TARANIS and SPHRAY are largely similar, though the latter contains significantly less full-neutral material. TARANIS also closely matches the results of RSPH, and we see that many of the other codes have noisy histograms at this resolution (20 bins); on reducing the bin-count (not shown), all ray-traced codes achieve similar distributions.

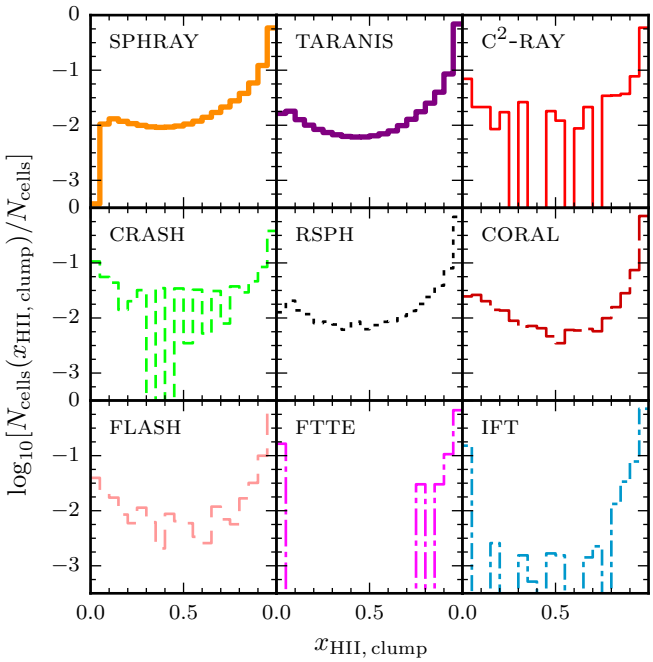


Figure 5.16. Test III, histograms of the fraction of in-clump cells with a given ionized fraction at $t = 15\,\mathrm{Myr}$.

The corresponding temperature distributions at early and late times can be seen in Figure 5.17 and Figure 5.18, respectively. At 1 Myr, TARANIS has a bimodal temperature distribution, with a secondary peak at a few hundred Kelvin; this feature is seen, to a lesser extent, in both CRASH and RSPH, but the lower clump temperatures in SPHRAY are more broadly distributed. This is consistent with its heating front being both wider and penetrating more deeply into the clump.

The low-temperature tail of the distributions is eroded over time, and all codes reproduce sharp peaks around $10^4\,\mathrm{K}$ by 15 Myr. For TARANIS and SPHRAY the peak is particularly sharp, likely a consequence of the relatively-higher resolution of the SPH dataset. The total temperature range is also narrow for these codes. Finally, we see also that the cooler region at the back of the clump in the TARANIS slice of Figure 5.15 constitutes only a small fraction of the total; it therefore appears that TARANIS reproduces a clump very similar to that of SPHRAY, with cooler regions present only in the central slice(s) already shown.

In closing, it should be noted that the test3-170-smooth dataset produced essentially identical results for otherwise-identical input parameters. Slightly more shot-noise is visible, but this is to be expected, since the ray count was not increased to maintain an equivalent sampling. On the other hand, the test3-128 dataset did result in notable

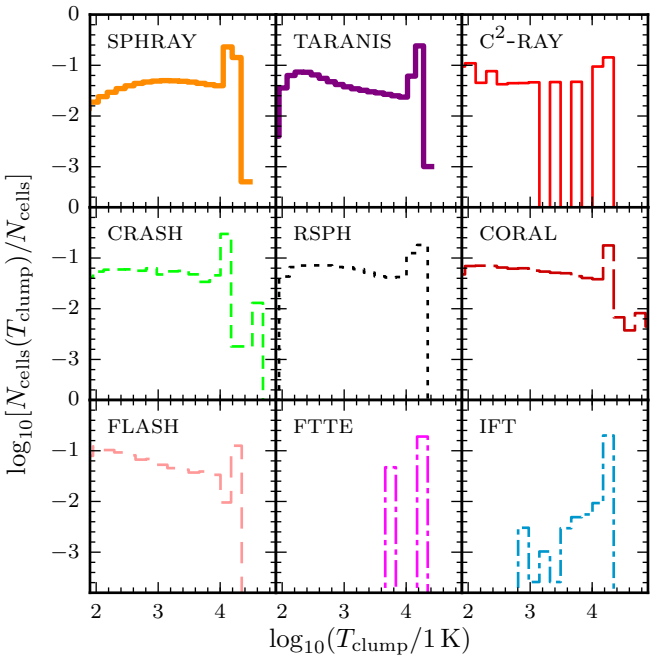


Figure 5.17. Test III, histograms of the fraction of in-clump cells with a given temperature at $t = 1\,\mathrm{Myr}$.

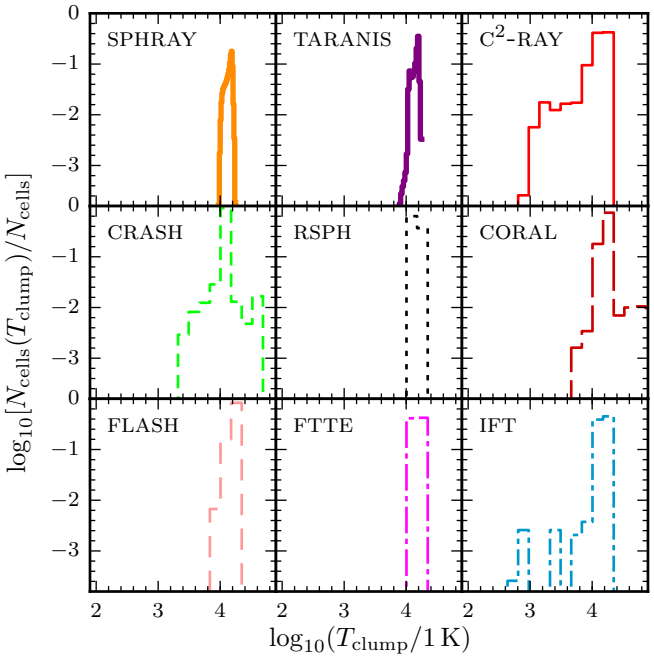


Figure 5.18. Test III, histograms of the fraction of in-clump cells with a given temperature at $t = 15\,\mathrm{Myr}$.

differences. Most evidently, neither the heating nor the ionization front penetrated as deeply into the clump. This was true for both TARANIS and SPHRAY, and to a similar degree: their $x_{\rm HI} - 0.5$ points are offset, source-ward, a comparable distance, occurring at $r \approx 0.78 L_{\rm box}$ at $t = 15\,{\rm Myr}$. The less-smooth SPH particle distribution will result in greater density variation, so clearly test3-128 is expected to be clumpier; on average, this results in higher recombination rates, which is entirely consistent with the above findings.

In any case, the differences between the two codes already discussed here apply to all three datasets. While some of these might be considered significant, the two codes respond similarly to changes in the initial conditions; in fact, the variations induced by the relatively minor changes here are comparable to the variations between the codes for any given dataset.

5.4 Test IV

Test IV is the most realistic test. Black body sources, of effective temperature $10^5\,{\rm K}$, are placed at the centres of each of the 16 most-massive haloes (density peaks) of a $z = 9$ cosmological simulation snapshot. The box has a comoving side-length $0.5h^{-1}$ cMpc, with the original simulation composed of a 128^3 grid and 2×64^3 particles, the former modelling the gas hydrodynamics. The temperature is set, initially, to 100 K everywhere. IL06 assume that each source lives for $t_s = 3\,{\rm Myr}$ and emits a total number of ionizing photons per hydrogen atom, $N_\gamma = 250$, over that time. Gas is taken to be composed entirely of hydrogen. Thus, for a source halo of total mass M, the ionizing photon count per second is

$$\dot{N}_\gamma = \frac{N_\gamma}{t_s} \frac{M}{m_p} \frac{\Omega_b}{\Omega_0},$$

where m_p is the proton mass, and cosmological constants $\Omega_0 = 0.27$, $\Omega_b = 0.043$ and $h = 0.7$ as adopted (as in IL06). It is assumed that all sources turn on at the beginning of the radiative transfer process, i.e. $z = 9$, and the system is evolved, under radiative-transfer only, for 0.4 Myr.

The 128^3 gas-density grid[7] and source positions and luminosities[8] were taken from the comparison project website.

[7] `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project/files/density.bin`, accessed 2017-07-18

[8] `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project/files/sources.dat`, accessed 2017-07-18

Initial conditions

The provided 128^3 cell uniform-grid density field must be converted into a SPH density field. In essence, this is achieved by placing into each cell i a number, n_i, of particles of mass m, where n_i is proportional to that cell's fractional density, $n_i \propto \rho_i / \sum_j \rho_j$. All particles have equal masses, set such that the total mass contained within the box, divided by its volume, is equal to $\Omega_b \rho_c$, where $\rho_c = 3H^2(z = 9)/\,8\pi G$ is the critical density at $z = 9$ and $\Omega_b = 0.043$ is the baryon density parameter.

In practice, this was achieved as follows. First, for convenience, the $128 \times 128 \times 128$ field is mapped to a linear array of length 128^3, and all elements divided by the total density. We then compute the cumulative sum of this fractional-density array. $N = 128^3$ uniform-random numbers in [0, 1) are then generated, and sorted into the cumulative density array. More specifically, the index at which each uniform-random value would be inserted into the cumulative density array is computed. This index corresponds to a particular cell in the original densities array, and hence to a particular cell position. Each index is thus converted to its corresponding cell location, and taken to be an SPH particle position; we thus have, approximately, the correct number of particles per cell. Each particle's co-ordinates are then dispersed randomly within their cell, easily achieved by adding a value xL_{cell} to each co-ordinate, where L_{cell} is the side-length of a cell, and x is a uniform-random value in $[-0.5, 0.5)$. Again, note that this is is a poor initial condition for hydrodynamical evolution, but sufficient for radiative transfer. Finally, the resulting particle distribution is passed to GADGET-2 as an initial condition file to compute smoothing lengths.

Results and discussion

Here, TARANIS traces 1024 rays per source per timestep, and uses a timestep multiplier of $f = 0.5$ (recall Section 4.5). However, to attain viable run times, a minimum timestep of 10^7 s was set, resulting in an effective $f \approx 5$ for some small number of particles (the actual computed timestep reached as low as 10^6 s). Essentially all computed timesteps were below this threshold, and hence increased to match it, casting doubt on the accuracy of the results. This is discussed further in Section 5.5.

Note that the SimpleX data for this test appears to contain cells with bad data; the corresponding cells are left white in this section. Further, the version of SimpleX

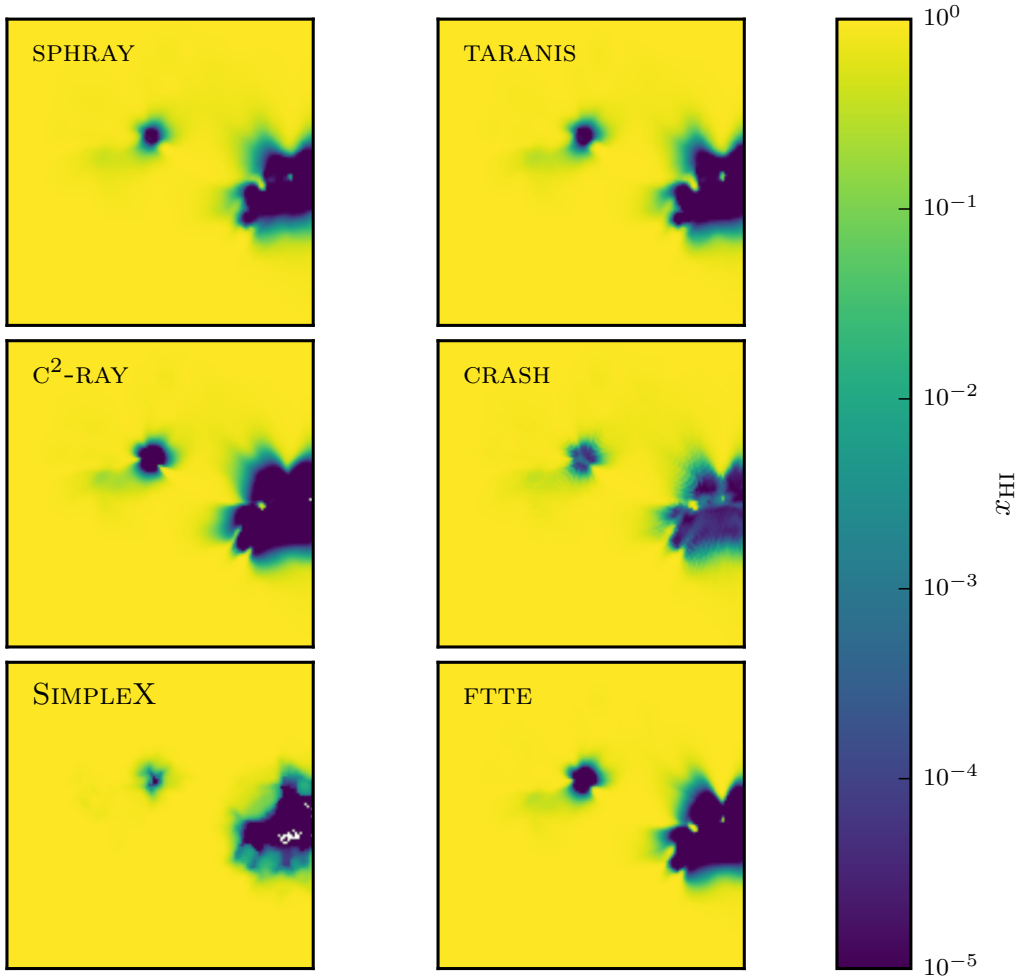


Figure 5.19. Test IV neutral-fraction slices at $t = 0.05\,\mathrm{Myr}$. Slices are one cell in depth, or $0.0039h^{-1}$ cMpc in all cases.

used for the CRTCP did not track the temperature state, and so is not included in temperature results.

In Figure 5.19, a slice in the x-y plane at $z = L_{\mathrm{Box}}/2$ is shown; specifically, cells with z-index $z_i = 63$ for $z_i \in [0, 128)$. For both SPHRAY and TARANIS, the SPH field is first interpolated onto a $128 \times 128 \times 128$ grid. TARANIS shows good agreement with all codes, and with SPHRAY in particular. The corresponding temperature slices are shown in Figure 5.20, and again TARANIS is generally in good agreement with the other codes. SPHRAY shows slightly more heating in the cooler regions, and more shot noise where TARANIS has smoother transitions, as has been seen in the previous tests.

These trends continue on to 0.4 Myr, as shown in Figures 5.21 and 5.22. Again, the ionization state produced by TARANIS most closely matches that of SPHRAY, but

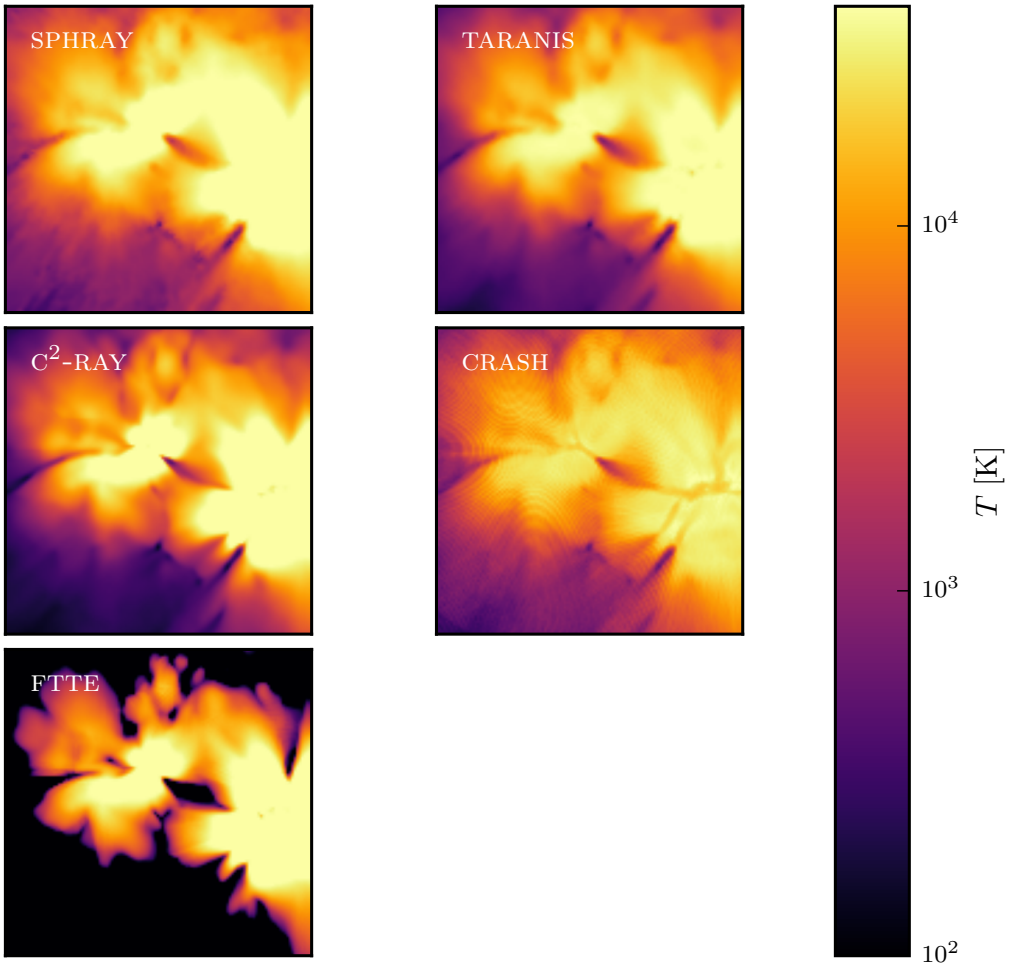


Figure 5.20. Test IV temperature slices at $t = 0.05\,\mathrm{Myr}$. Slices are one cell in depth, or $0.0039h^{-1}$ cMpc in all cases.

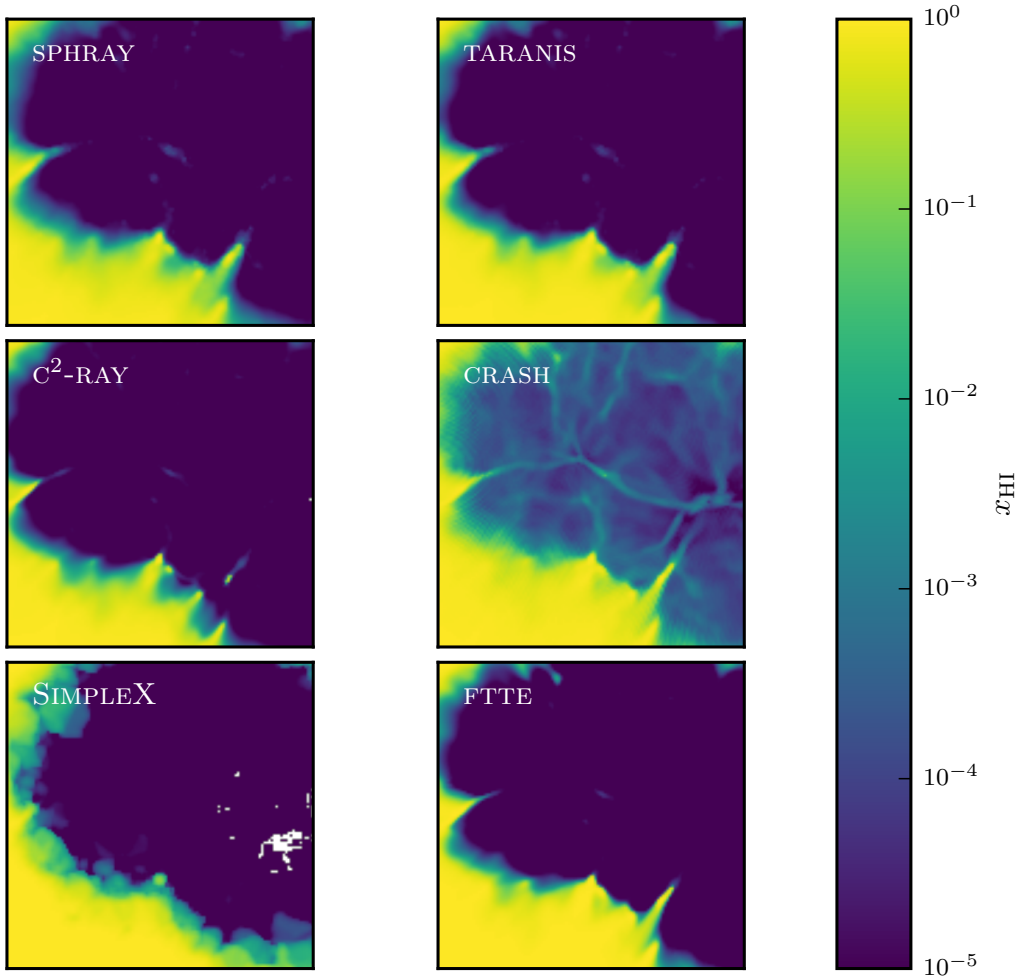


Figure 5.21. Test IV neutral-fraction slices at $t = 0.4\,\mathrm{Myr}$. Slices are one cell in depth, or $0.0039h^{-1}$ cMpc in all cases.

all codes are in relatively good agreement regarding the overall morphology, size and extent of the ionized region. Most codes also agree that the entire box-slice has, nearly, reached temperature equilibrium by this time, and again we see the heating-front of SPHRAY extend slightly farther out, with TARANIS lying between it and C^2-RAY.

The volume- and mass-weighted ionization fractions are given in Figure 5.23. These corroborate the similar ionization structures seen previously, with all codes agreeing to within about 10%. Note also the cross-over in the volume- and mass-weighted means, consistent with an inside-out reionization model: at early times, high-mass regions are ionized first by the sources they contain, while at late times the large, low-mass voids also become ionized.

Histograms of the neutral fraction are shown in Figure 5.24. All codes are very

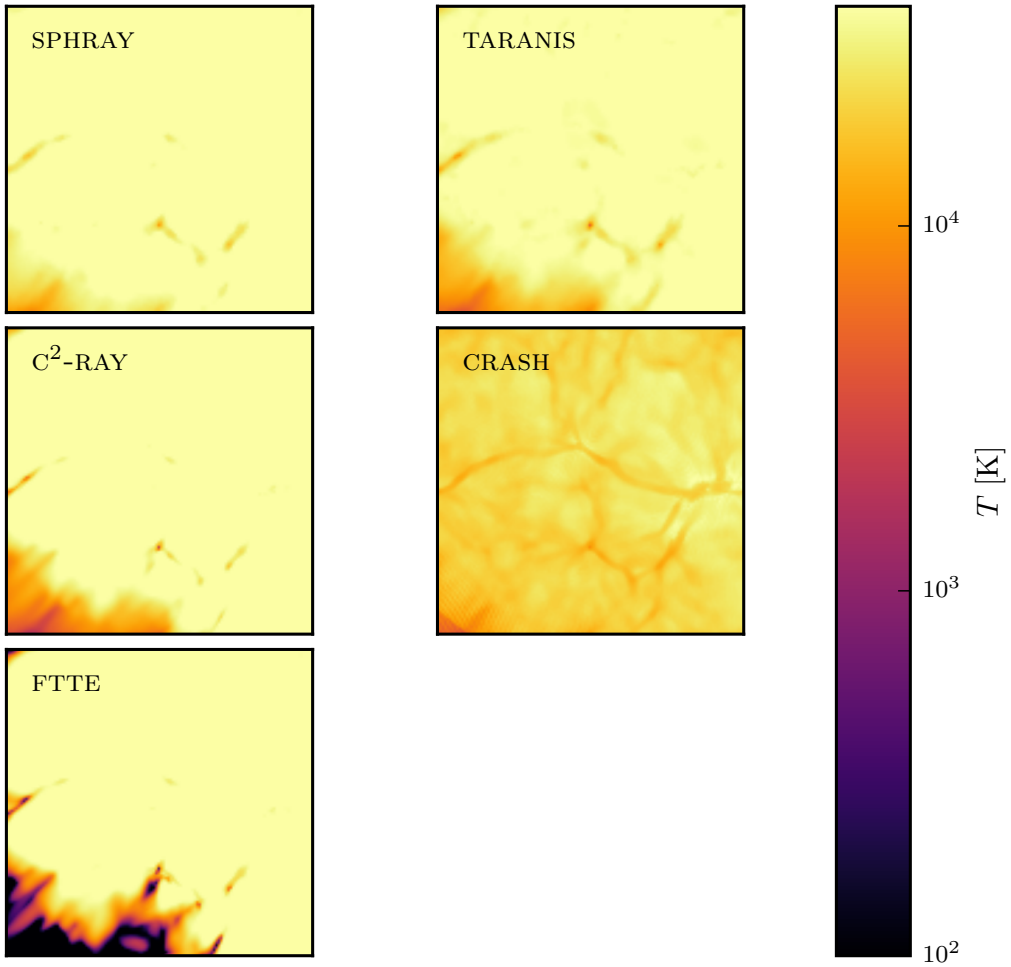


Figure 5.22. Test IV temperature slices at $t = 0.4\,\mathrm{Myr}$. Slices are one cell in depth, or $0.0039h^{-1}$ cMpc in all cases.

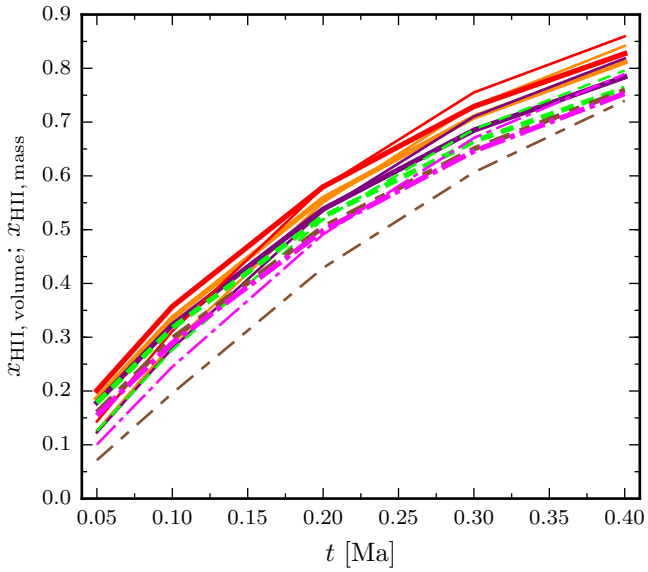
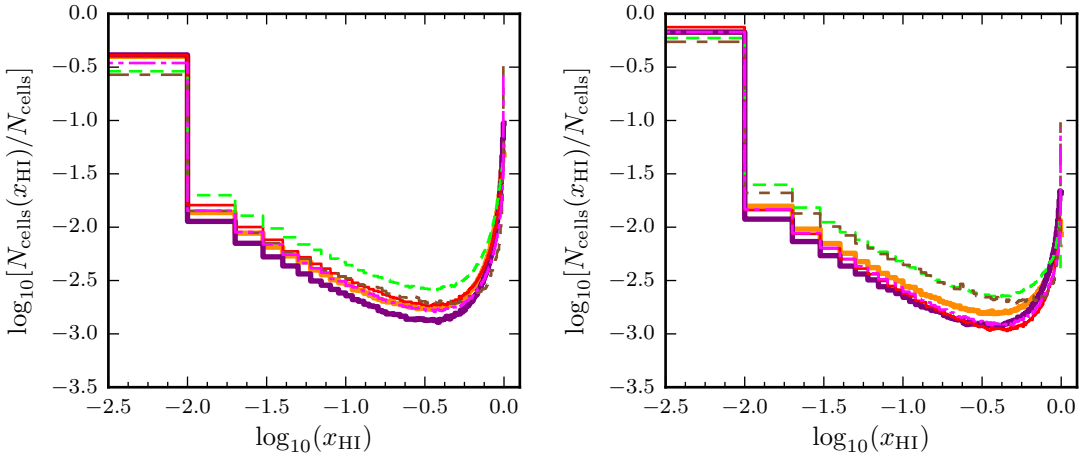


Figure 5.23. Test IV volume-weighted (thin) and mass-weighted (thick) ionization fractions over time.

consistent at the earliest times (not shown), except at values $x_{\mathrm{HI}} \lesssim 10^{-2}$, though here TARANIS remains identical to SPHRAY. At intermediate times, TARANIS tracks the state of the other codes slightly less closely than SPHRAY, generally predicting fewer cells for any given neutral fraction $x_{\mathrm{HI}} \lesssim 10^{-0.25}$, but with a slightly larger peak in the number of $x_{\mathrm{HI}} \sim 1$ cells. At late times, TARANIS and the other CRTCP codes agree more closely in the ionized regions, and we see further evidence of wider ionization fronts in SPHRAY in the form of an increased number of cells at $x_{\mathrm{HI}} \sim 10^{-0.5}$. TARANIS still retains a sharper peak at the fully-neutral end, which may be a symptom of an insufficient number of rays, or too large timesteps, and hence slower ionization fronts. In any case, general agreement between TARANIS and most other codes is clear.

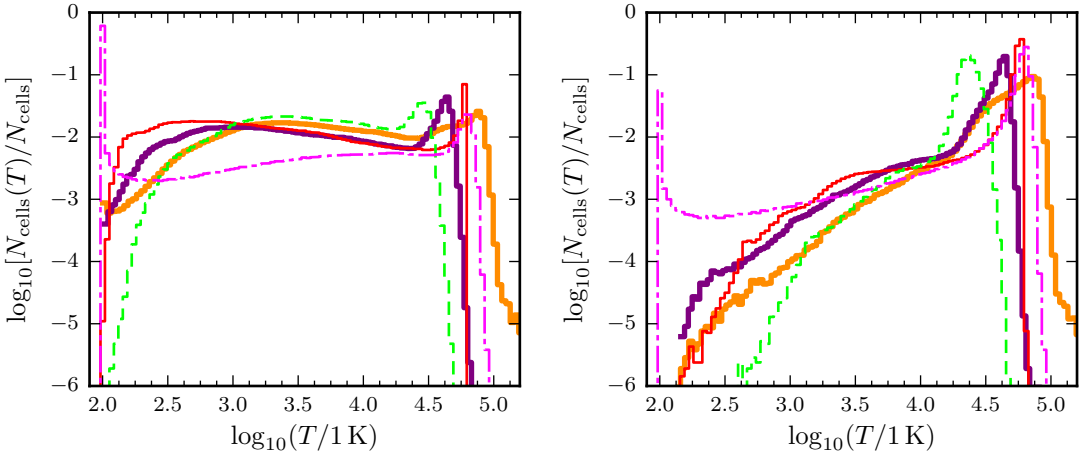
Even at early times, the temperature distribution is not so consistent, as seen in Figure 5.25. The range of peak temperatures spans a factor of a few, with TARANIS toward the lower end; however, its temperature peak is broader than that of the other CRTCP codes, with the sharp post-peak drop still being consistent. We also see slightly more particles at the lowest temperatures in TARANIS, a feature preserved across all snapshots. These lower temperatures are consistent with, and a possible cause of, the higher neutral fractions seen in the TARANIS results. Note that FTTE as in IL06 does not sample high-energy photons, hence the lower-temperature peak.

While again the general features produced by all codes are the same, and there is



(a) $t = 0.2\,\mathrm{Myr}$. **(b)** $t = 0.4\,\mathrm{Myr}$.

Figure 5.24. Test IV, histograms of the fraction of cells with a given neutral fraction.



(a) $t = 0.05\,\mathrm{Myr}$. **(b)** $t = 0.4\,\mathrm{Myr}$.

Figure 5.25. Test IV, histograms of the fraction of cells with a given temperature.

no immediate cause for concern, differences are most apparent in these temperature distributions. Changes here, for example if these results are not well-converged, will likely have a noticeable effect on the ionization state.

IL06 also run a modified version of test IV with softer spectra in an effort to reduce the differences due primarily to high-energy photons. There they find that this produces more-similar temperature distributions between C^2-RAY and CRASH; repeating this modified test with both TARANIS and SPHRAY may also be enlightening, issues of convergence notwithstanding.

5.5 Convergence

It was stated previously that tests I through III were converged, with the first two tracing 4096 rays per timestep and using a timestep multiplier of $f = 0.2$, and the third tracing 1024 rays per timestep with $f = 0.5$. In all cases, four sample points are used for the Gauss-Legendre integrals over $[\nu_0^{\mathrm{HI}}, \nu_0^{\mathrm{HeI}})$ and $[\nu_0^{\mathrm{HeI}}, \nu_0^{\mathrm{HeII}})$ and eight for the Gauss-Laguerre integral over $[\nu_0^{\mathrm{HeII}}, \infty)$. In this section, the degree to which the above-reported results may be considered converged is discussed, along with the possibly non-converged test IV results.

Expected ray count requirement

First, let us derive the expected number of rays required to sufficiently sample a uniform SPH field. Consider the cross-sectional area, A, of a particle of radius Δr,

$$A = \pi \Delta r^2.$$

If placed a distance l from the source, and provided $l \gg \Delta r$, the solid angle subtended by this particle, Ω, is defined by

$$\Omega l^2 = \pi \Delta r^2.$$

The total number of rays needed, on average, to sample particles is then $4\pi/\Omega$. To be conservative, let us take $\Delta r = L_{\mathrm{box}}/N^{1/3}$, where N is the number of particles. We could instead choose the smoothing volume, if we merely wanted to ensure all particles were intersected, on average. But, recall from Section 4.3 that we take photoionization rates to be constant over a spatial scale $\Delta r < h_{\mathrm{smooth}}$. For test II, some relevant values are given in Table 5.6.

Table 5.6. The number of rays needed to, on average, correctly sample all particles a distance l from a point-source, $N_{\rm rays}(l,\,\Delta r)$, and to merely intersect them, $N_{\rm rays}(l,\,\langle h_{\rm smooth}\rangle)$.

l	$N_{\rm rays}(l,\,\Delta r)$	$N_{\rm rays}(l,\,\langle h_{\rm smooth}\rangle)$
$r_s = 5.4\,\mathrm{kpc}$	1.1×10^4	2.9×10^3
$L_{\rm box} = 6.6\,\mathrm{kpc}$	1.6×10^4	4.3×10^3
$l_{\rm max} = \sqrt{3}L_{\rm box}$	4.9×10^4	1.3×10^4

Applying a somewhat similar argument for test III, and noting that the plane-parallel ray distribution of GRACE aims to be uniform across the box face, rather than optimally denser through the clump, one finds $N_{\rm rays} = L_{\rm box}^2/\pi\Delta r^2 = (200 \times N)^{2/3}/\,\pi = 1.78 \times 10^5$ is necessary for $N = 128^3$.

So it appears now that tests I to III may in fact not be correctly sampling particles — though we still expect almost all particles to be intersected each timestep in the first two. This latter point is important, as in TARANIS a non-intersected particle will have zero ionization and heating rates, but non-zero recombination and cooling rates. Indeed, some shot noise was noted in Sections 5.1 to 5.3.

Test II

The convergence of test II, for various values of $N_{\rm rays}$ is shown here, with the corresponding key below, in Figure 5.26. Similar results hold for test I, though for brevity these are not shown.

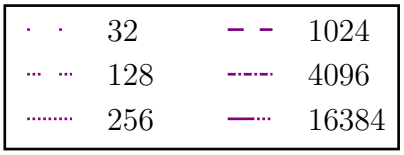


Figure 5.26. $N_{\rm rays}$ per timestep test II results legend.

In Figures 5.27 and 5.28, shot noise is clearly visible at low ray counts, but not apparent when the number of rays is equal to that predicted for correct sampling.

We see in Figure 5.29 that this remains visible in the neutral fraction for ray counts $\lesssim 10^3$ after several times the recombination time, $t_{\rm rec} \sim 120\,\mathrm{Myr}$. Temperatures, in Figure 5.30, are not so strongly affected.

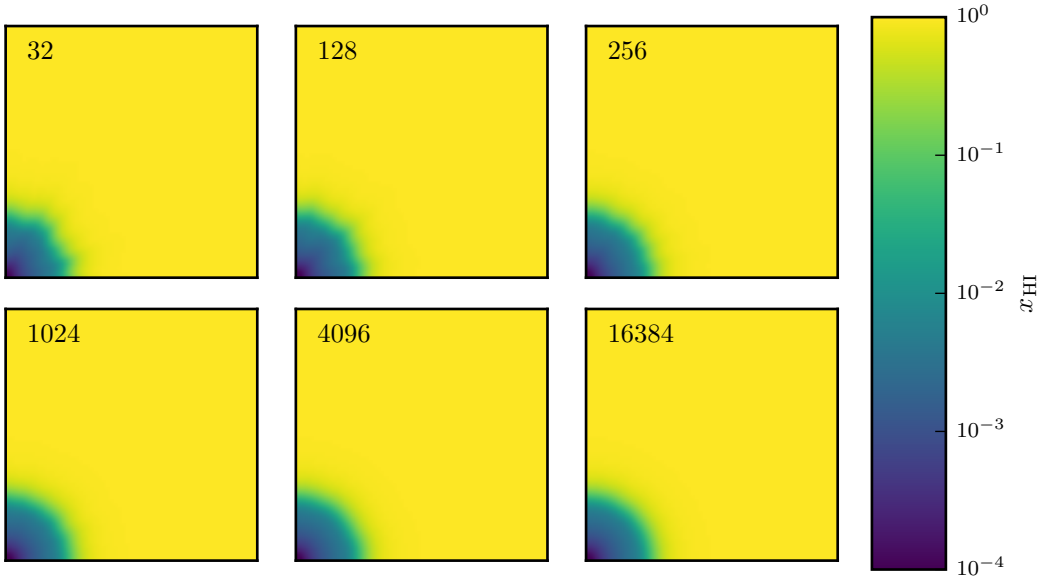


Figure 5.27. Test II neutral-fraction slices at $t = 10\,\mathrm{Myr}$ for various values of N_{rays} per timestep.

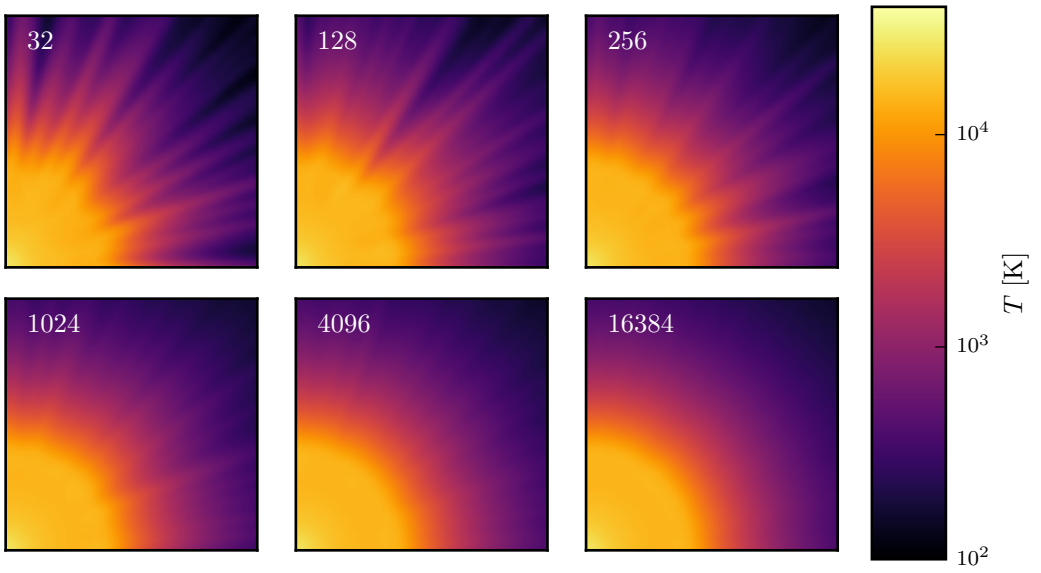


Figure 5.28. Test II temperature slices at $t = 10\,\mathrm{Myr}$ for various values of N_{rays} per timestep.

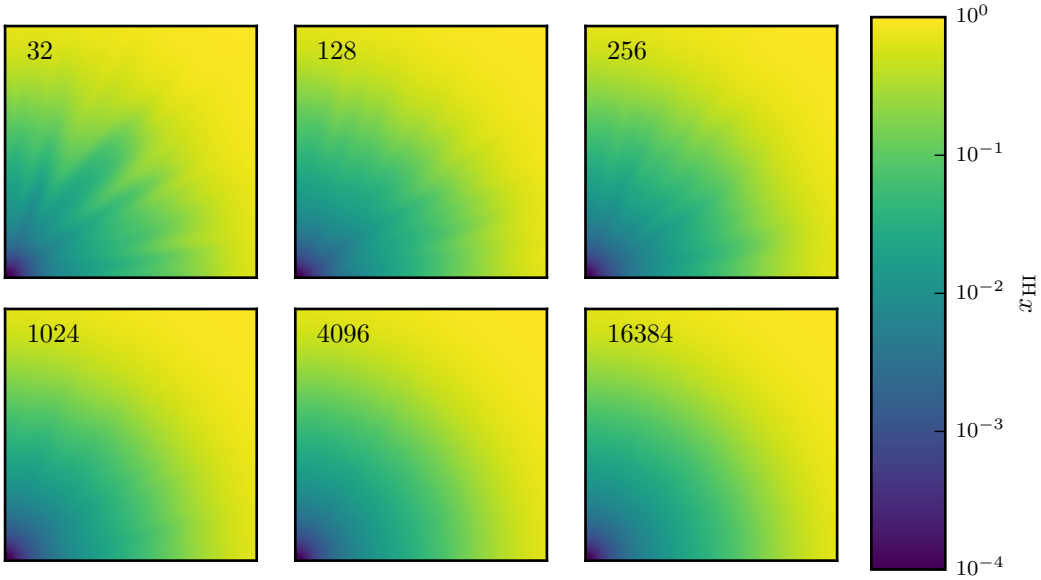


Figure 5.29. Test II neutral-fraction slices at $t = 500\,\mathrm{Myr}$ for various values of N_{rays} per timestep.

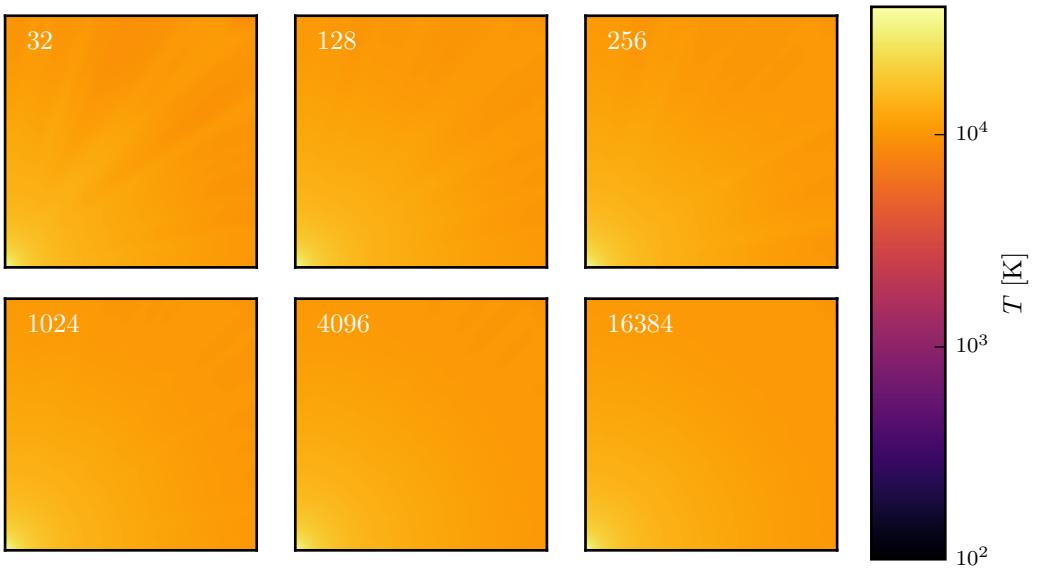
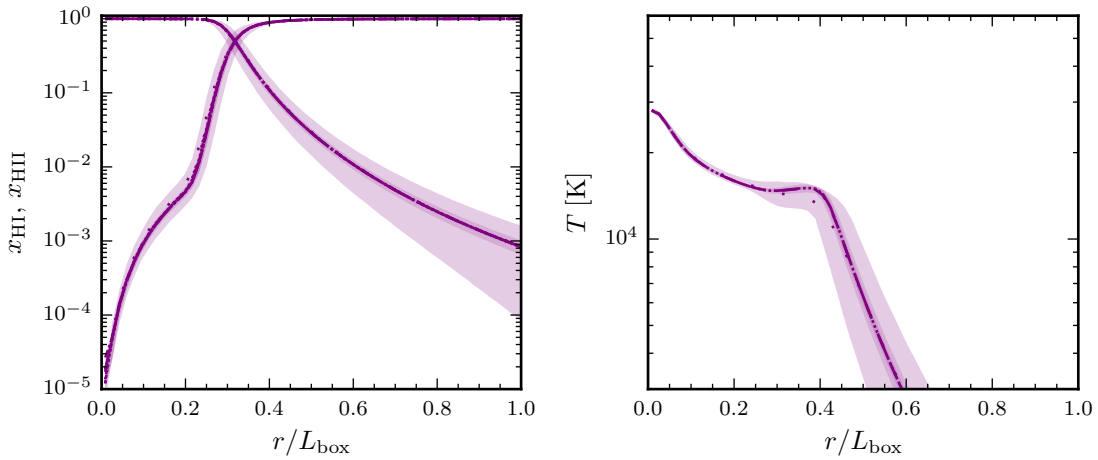
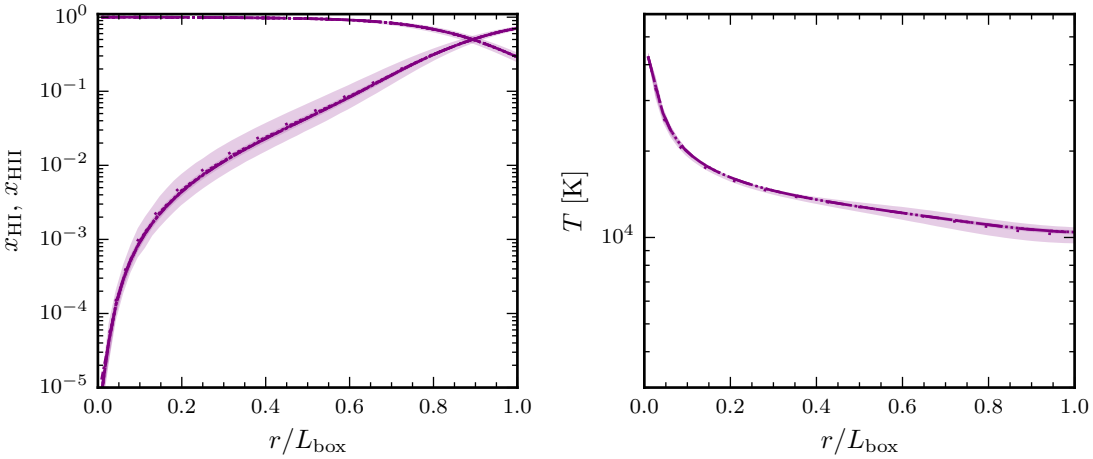


Figure 5.30. Test II temperature slices at $t = 500\,\mathrm{Myr}$ for various values of N_{rays} per timestep.



(a) x_{H} profiles at $t = 10\,\mathrm{Myr}$.

(b) T profiles at $t = 10\,\mathrm{Myr}$.



(c) x_{H} profiles at $t = 500\,\mathrm{Myr}$.

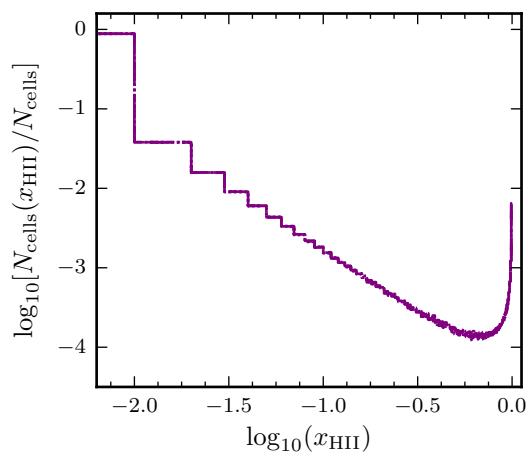
(d) T profiles at $t = 500\,\mathrm{Myr}$.

Figure 5.31. Test II radially-averaged ionization fraction and temperature profiles at various values of N_{rays} per timestep. Shaded regions denote approximate 1σ bounds for N_{rays} of 32, 1024 and 16 384, the latter being smaller than the linewidth and thus not clearly visible.

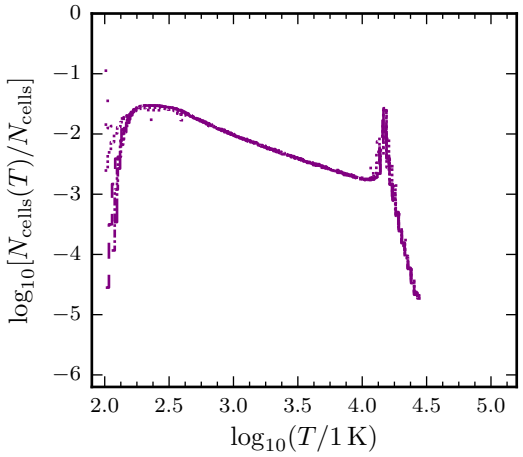
The radially-averaged profiles of Figure 5.31 are visually identical at all but the lowest ray counts. To better quantify the above-noted shot noise, in Figure 5.31 scatter in the particle state at each radial bin is also shown. Specifically, the shaded regions are bounded by the (linearly-interpolated) 16th- and 84th-percentiles, encompassing 68% and centred about the median, and given for 32, 1024 and 16 384 values of N_{rays}. (Scatter for the latter is typically less than the profile line-width, and may not be visible.) Radial asymmetry is clearly low for $N_{\mathrm{rays}} \geq 1024$.

Finally, ionized-fraction and temperature histograms are shown in Figure 5.32. Again, results are clearly converged for all but the lowest ray counts.

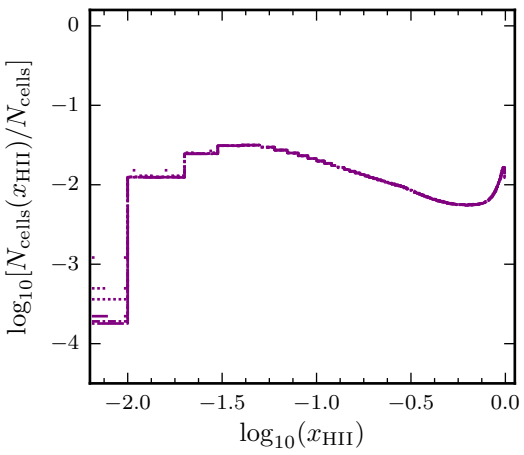
A run with 4096 rays and a timestep factor increased to $f = 0.5$ (from $f = 0.2$) was



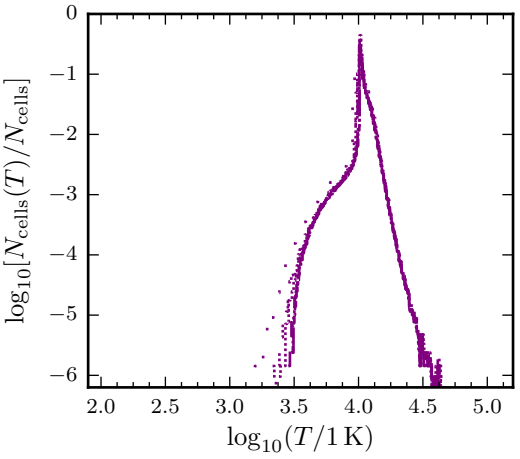
(a) x_{HII} histograms at $t = 10\,\mathrm{Myr}$.



(b) T histograms at $t = 10\,\mathrm{Myr}$.



(c) x_{HII} histograms at $t = 500\,\mathrm{Myr}$.



(d) T histograms at $t = 500\,\mathrm{Myr}$.

Figure 5.32. Test II, histograms of the fraction of cells with a given ionization fraction or temperature at various values of N_{rays} per timestep.

also made. All results were found to be essentially indistinguishable, excepting some noise in both datasets at the lowest fractional cell-counts of the histograms, totalling a negligible number of particles. The total runtime at $f = 0.5$ was 70% of the runtime at $f = 0.2$, however, so the computed heating- and/or ionization-timescales were decreased.

Finally, doubling the number of Gaussian quadrature sample points from 4 (Gauss-Legendre) and 8 (Gauss-Laguerre) to 8 and 16 did not have any visible impact.

Test III

Test III used the lower value of 1024 rays per timestep, and an increased timestep multiplier of $f = 0.5$. Results for several values of $N_{\rm rays}$ and f are given here, with the key below. For the case $N_{\rm rays} = 16\,384$, rays are incident from an area $(2r_{\rm clump} + 4\langle h_{\rm out}\rangle)^2 = (2\,{\rm kpc})^2$, centred on the clump in the y-z plane, where $\langle h_{\rm out}\rangle \approx 0.1\,{\rm kpc}$ is the approximate mean smoothing length of particles outside the clump. This is sufficient to correctly sample particles within the clump, but avoids the significant cost of applying this ray-density over the entire box face (which would require $N_{\rm rays} \approx 178\,000$).

Figure 5.33. $N_{\rm rays}$ and timestep factor test III results legend.

In Figures 5.34 and 5.35, shot noise is clearly reduced at higher ray counts, and also at smaller values of f, where the latter results in a greater number of rays traced over a given simulation time. While only intermediate times are shown, this is seen over all snapshots. However, no significant differences are otherwise apparent.

Some minor differences in the ionization-state profiles are present at all times, becoming most apparent at $t \gtrsim 5\,{\rm Myr}$. Slightly more ionization is seen in the source-side region of the clump for $x_{\rm HI} < 0.5$. This is shown at late times in Figure 5.36, and compared with the RABACUS equilibrium results. While this results in a closer profile in the source-side of the clump, the difference is small, and very little change is seen in the still-shielded region where the codes differ most significantly. Temperature results

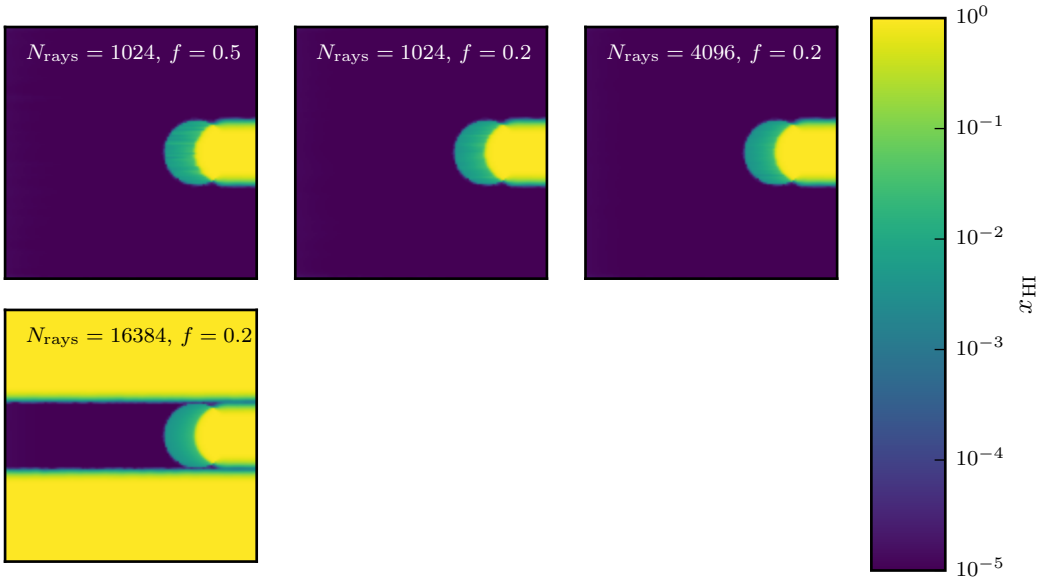


Figure 5.34. Test III neutral-fraction slices at $t = 5\,\mathrm{Myr}$ for various values of N_{rays} and f.

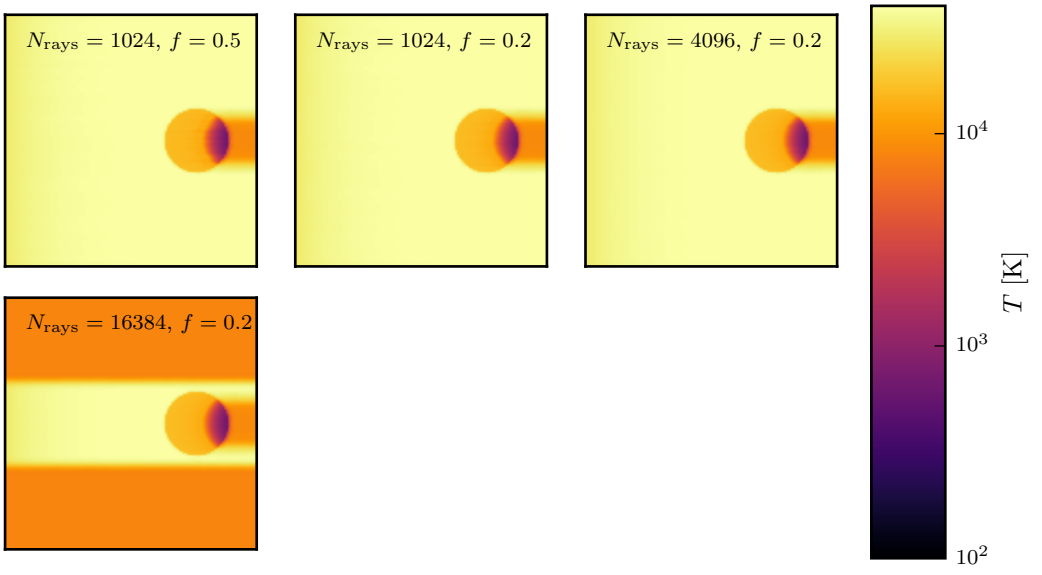
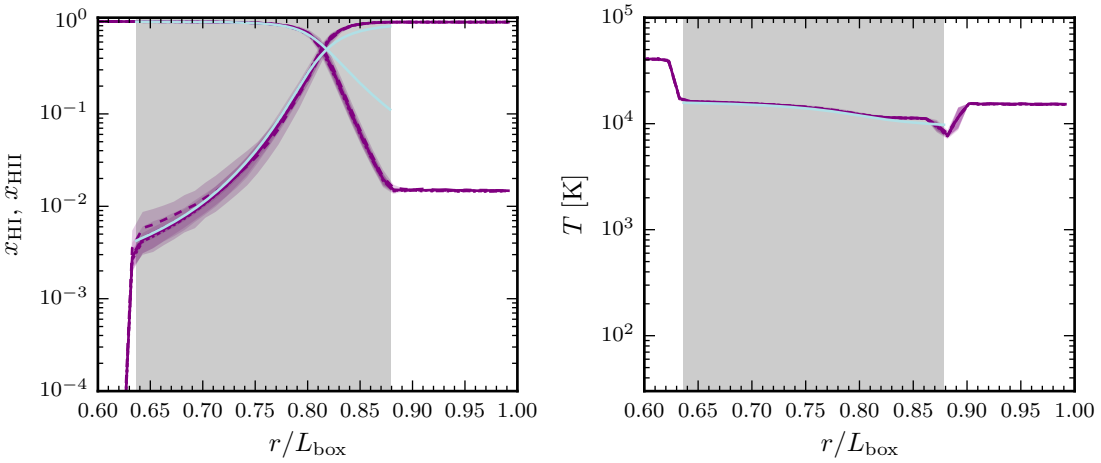


Figure 5.35. Test III temperature slices at $t = 5\,\mathrm{Myr}$ for various values of N_{rays} and f.



(a) x_{H} profiles at $t = 15\,\mathrm{Myr}$.

(b) T profiles at $t = 15\,\mathrm{Myr}$.

Figure 5.36. Test III radially-averaged ionization fraction and temperature profiles at various values of N_{rays} and f. The grey-shaded region denotes the clump, and the purple-shaded regions denote approximate 1σ bounds for all profiles.

are virtually identical for all TARANIS runs. As for test II, approximate 1σ bounds are shown as purple-shaded regions, given for all four variations here tested for convergence. (Scatter for the $N_{\mathrm{rays}} = 16\,384$ run is typically less than the profile line-width, and may not be visible.) Scatter in the ionization state remains moderate at the source-side clump boundary even up to $N_{\mathrm{rays}} = 4096$, $f = 0.2$, though is negligible for the temperature state.

Ionized-fraction histograms for all particles within the clump are shown in at intermediate times in Figure 5.37. No differences are observed, which holds also at early and late times. The corresponding temperature histogram is given in Figure 5.38. Slightly fewer particles exist at lower temperatures for higher ray counts, though by late times (not shown) the distributions are extremely similar.

While not necessarily as well-converged as the results presented for tests I and II, it is clear that the TARANIS results for test III are sufficiently similar to their converged solutions. This is despite the seemingly too-low value of N_{rays}.

Test IV

Test IV makes clear a significant, and unresolved, problem with TARANIS. As noted above, for the 128^3 dataset described in Section 5.4 the computed timestep was $< 10^7\,\mathrm{s} \sim 0.3\,\mathrm{yr}$ for essentially all steps, which is extremely small. A minimum-allowed timestep of $10^7\,\mathrm{s}$ was therefore set, but in total, this still required 58 days of wall-clock time for

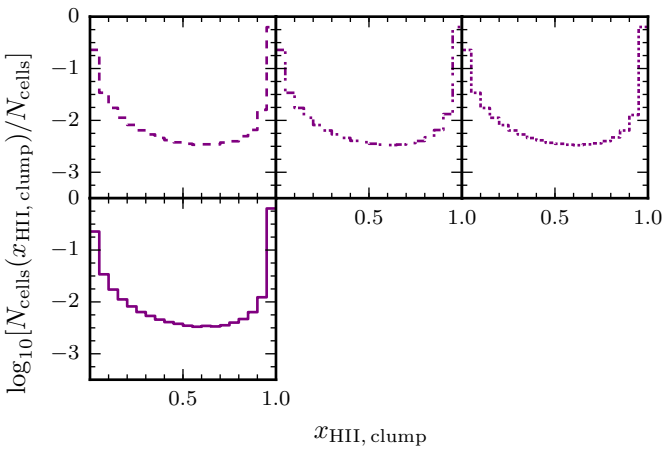


Figure 5.37. Test III, histograms of the fraction of cells with a given ionization fraction at various values of N_{rays} and f at $t = 5\,\mathrm{Myr}$.

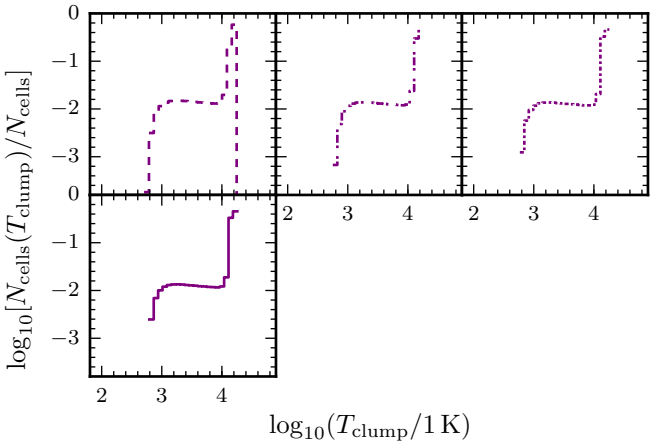


Figure 5.38. Test III, histograms of the fraction of cells with a given temperature at various values of N_{rays} and f at $t = 5\,\mathrm{Myr}$.

completion. Convergence tests, as above, are therefore impractical. Possible solutions to this issue are presented in Section 5.7.

In order to obtain some useful information regarding the validity of these test results, a $\sim 64^3$ particle dataset was generated. Further, in an effort to improve the quality of this initial condition, the method suggested by AL08 was used.

We begin with a glass-like SPH field of N particles with a uniform density equal to the peak density of the target distribution, ρ_{peak}. Recall that the target distribution is itself specified as N_{grid} cells. Particles are then mapped to their corresponding cells in this reference file according to the positions of their centres; thus a target density for each particle is obtained. As the dataset is uniform, we expect each cell to contain approximately $N_{\text{cell}} = N/N_{\text{grid}}$ particles. Then, for each particle, we generate a uniform random variable $\rho_c \in [0, \rho_{\text{peak}})$ and remove that particle from the glass-like dataset if $\rho_c > \rho_i$, where ρ_i is the density of the the particle's reference cell. Contrary to the method described in Section 5.4, which is noisy everywhere, here the high-density regions should retain their glass-like properties. Smoothing lengths are computed by the GADGET-2 code.

Note that the expected number of final particles is

$$N' = \frac{N}{N_{\text{grid}}} \frac{1}{\rho_{\text{peak}}} \sum_i \rho_i$$

which, for $N' = 128^3$, requires an input glass of $N \simeq 1.15 \times 10^9$ particles. No such glass was available, hence why this method was not used in Section 5.4 (it would, however, be trivial to create one by tiling a smaller glass distribution). For $N = 512^3$, which is available, we obtain $N' = 2.5 \times 10^5 \approx 64^3$.

The reduced particle count results in a substantially more tractable problem. Now, a value $N_{\text{rays}} = 1024$ is comparable to the number of rays required for correct sampling. Further, the restriction on the timestep may be lowered, and indeed the minimum timestep specified is never actually reached. On this basis, we assume the results for the $\sim 64^3$ dataset to be relatively well converged, and compare them to those presented in Section 5.4. The key for the two input datasets is given below.

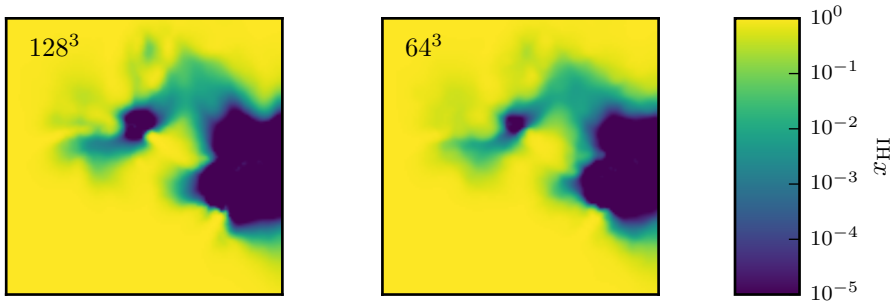


Figure 5.40. Test IV neutral-fraction slices at $t = 0.1\,\mathrm{Myr}$ for different initial condition datasets.

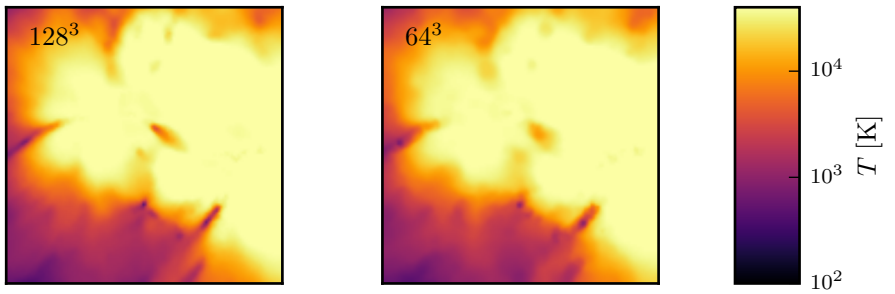


Figure 5.41. Test IV temperature slices at $t = 0.1\,\mathrm{Myr}$ for different initial condition datasets.

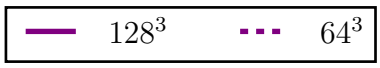


Figure 5.39. Test IV performance results legend.

Slices of the ionization and temperature state at early times are shown in Figures 5.40 and 5.41. While clearly variable, results are broadly similar. Differences are most noticeable in the ionization state, where ionized regions are marginally smaller for the 64^3 dataset, and their edges are not as well-defined. This smoothness is likely entirely attributable to the reduced spatial resolution. Further, SPHRAY also produces smaller ionized regions at early times with this dataset (not shown), and is similarly smoother. Slices are similar at late times, as shown in Figures 5.42 and 5.43.

Excellent agreement is seen in the neutral-fraction histograms, as shown in Figure 5.44.

Temperature results, in Figure 5.45, show some minor deviation at late times in the cooler regions, possibly indicative of under-sampling, or excess heating due to too-large timesteps, for the 128^3 dataset; conversely, since cool regions are likely to be distant from a source and thus of lower density, this may be due to the noisy, low spatial

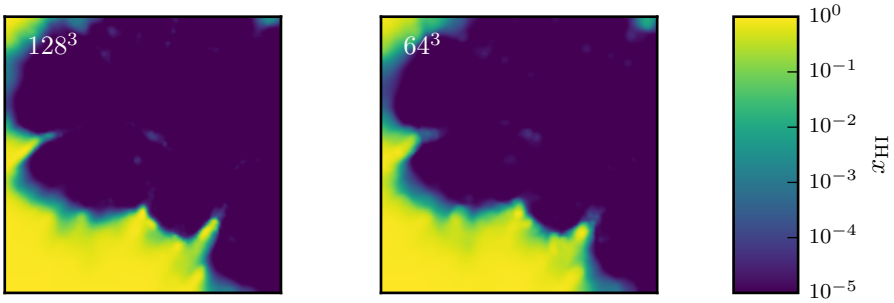


Figure 5.42. Test IV neutral-fraction slices at $t = 0.4\,\mathrm{Myr}$ for different initial condition datasets.

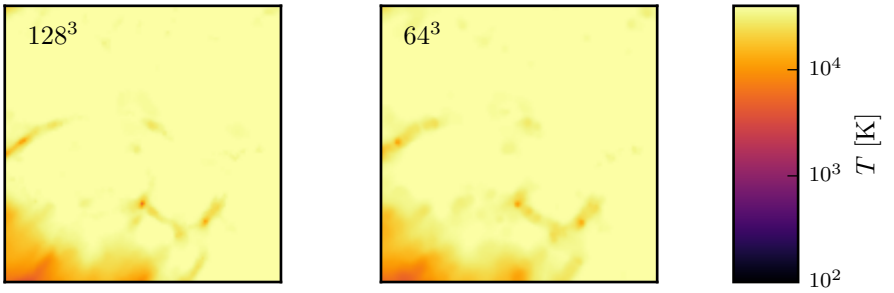
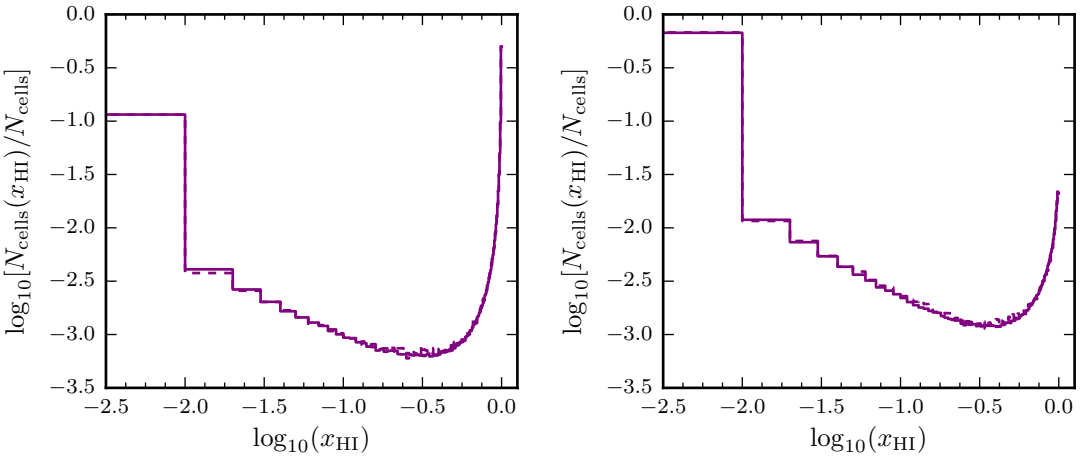


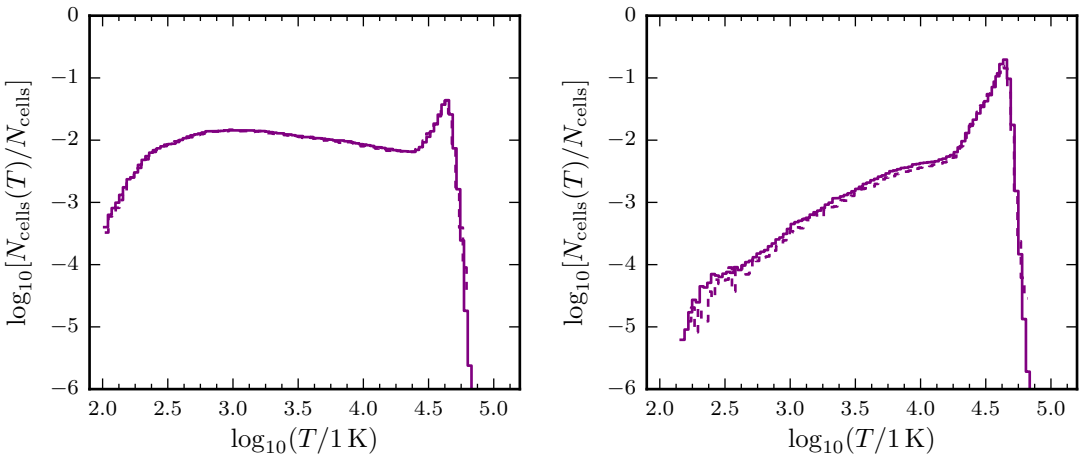
Figure 5.43. Test IV temperature slices at $t = 0.4\,\mathrm{Myr}$ for different initial condition datasets.



(a) $t = 0.05\,\mathrm{Myr}$.

(b) $t = 0.4\,\mathrm{Myr}$.

Figure 5.44. Test IV, histograms of the fraction of cells with a given neutral fraction for different initial condition datasets.



(a) $t = 0.05\,\mathrm{Myr}$. **(b)** $t = 0.4\,\mathrm{Myr}$.

Figure 5.45. Test IV, histograms of the fraction of cells with a given temperature for different initial condition datasets.

resolution achieved there in the 64^3 particle dataset. SPHRAY histograms are also noisy in this region (not shown). Finally, recall from Figure 5.25 that all codes were relatively inconsistent in this metric, far more so than the differences seen here.

In summary, while the convergence analysis performed here is substantially more limited than those of the other tests, we do not see any evidence that significant differences would arise were it feasible to run the 128^3 dataset with a greater number of rays per timestep, with a smaller timestep multiplier, or with a reduced minimum timestep. In fact, differences between TARANIS and SPHRAY noted in Section 5.4 apply equally well to this dataset (again, SPHRAY results are not shown).

5.6 Performance

With the validity of most of the TARANIS results and chosen parameters now demonstrated, we turn to its performance. The CRTCP tests are again used here; the data are available and, by design, they cover a range of benchmark problems. Test I is not covered, since isothermal radiative transfer is not expected to be of particular relevance to potential TARANIS users.

As a reference, runtimes are compared to those of SPHRAY. The effects of changes in N_{rays} per timestep and the size of the dataset are also explored.

All TARANIS results reported here were run on an NVIDIA Tesla M2090 GPU, and all SPHRAY results on an Intel Xeon E5-1620. The former was released in June 2012,

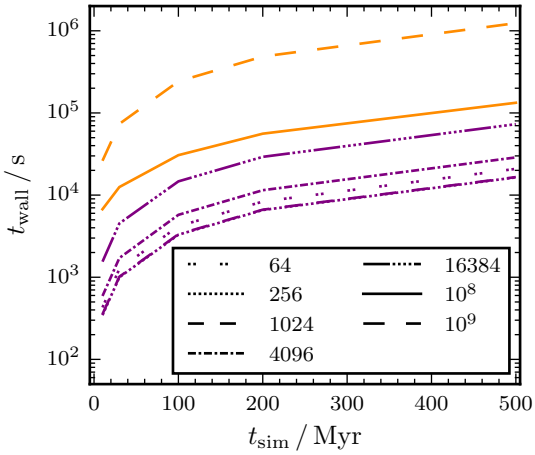


Figure 5.46. CRTCP test II wall-clock time as a function of simulation time for TARANIS (purple) and SPHRAY (orange), with varying ray counts. For TARANIS, values in the legend refer to N_{rays}, the number of rays per timestep; for SPHRAY, they refer to the total number of rays.

and is the most powerful Tesla card of its architecture (Fermi). The latter was released in Q3 2013, and has the highest clock speed of all processors of its architecture (Ivy Bridge-E), and hence for single-threaded applications should be considered the most powerful. Both the Tesla and Ivy Bridge-E lines formed, at their time of release, each manufacture's server-class offerings. Thus, while a truly objective comparison between CPU and GPU codes is not possible, the results here should be fairly representative of expected performance differences.

Test II

The wall-clock time as a function of simulation time is shown in Figure 5.46. For TARANIS, the value of N_{rays}, the number of rays traced per timestep, is modified; for SPHRAY, instead the total number of rays is varied. While not necessarily clear due to the scale, all lines are essentially linear. We see that, for ray counts shown to be converged ($N_{\mathrm{rays}} \gtrsim 1024$), TARANIS out-performs SPHRAY by almost an order of magnitude. For context, SPHRAY results at 10^8 rays are converged, while 10^7 are not; it is therefore likely that equivalent results can be achieved at a slightly reduced ray count and runtime. While initially encouraging, this actually suggests that a parallelized version of SPHRAY (the algorithm was noted in Chapter 2, Section 4.3 to scale usefully to a few 10s of cores) would be more-or-less comparable to TARANIS in runtime.

For TARANIS, relationship between total wall-clock time and N_{rays} shows some peculiarities. First, note that there is a 'sweet spot' for performance around $N_{\mathrm{rays}} = 1024$,

Table 5.7. CRTCP test II, fractional contributions to the total runtime for the ray-tracing and photoionization (and photoheating) computations in TARANIS, with varying ray counts per timestep.

N_{rays}	Fraction of runtime (%)	
	Tracing	Rates
64	46	1.8
256	41	6.7
1024	30	22
4096	16	46
16 384	12	65

and lower values actually increase the total runtime. One contributing factor is that, currently, TARANIS does not make optimal use of GRACE. Only N_{rays} rays are traced at a time, which was shown in Chapter 4, Section 9.5 to be sub-optimal for $N_{\text{rays}} \lesssim 1 \times 10^4$. This is an implementation detail, and in principle could be resolved by tracing rays from multiple sources simultaneously, or, appropriate here, by tracing rays for several timesteps simultaneously, and storing them.

In all cases, the runtime is dominated by the ray tracing and rate-computation (photoheating and photoionization) phases. Their fractional contributions to the total runtime are given in Table 5.7. For high ray counts, rate computation is solely dominant. For ray counts $\sim 10^3$, and again referring to Chapter 4, Section 9.5, specifically Figure 4.21, a modification such as that suggested above would reduce the total ray tracing time by a factor of ~ 4. Thus, for 1024 rays, rate computation would become dominant. For 256 rays, ray tracing and rate computation would be comparable. Finally, for the very low 64 rays, ray tracing would remain dominant.

Further investigation shows that the number of rays traced per timestep has an appreciable impact on the size of each timestep. In Figure 5.47, the distribution of timesteps taken over the full 500 Myr is shown. The smallest timesteps are all on the order of 10^{-3} Myr, and while most simulations peak in this region, higher ray counts increase the upper-limit of the timestep size by an order of magnitude. The distribution appears stable beyond the ray count required for all-particle sampling ($N_{\text{rays}} \approx 1.6 \times 10^4$), and close to stable where we expect most particles to at least be intersected. That particles intermittently receiving ionizing photons results in smaller timesteps is perhaps not surprising, as it will likely cause their ionization states to

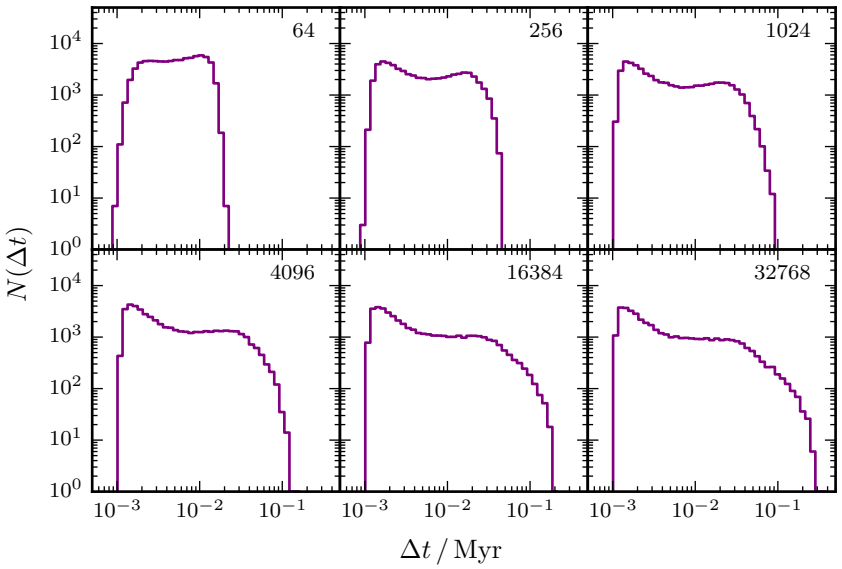


Figure 5.47. CRTCP test II histograms of timestep size for various values of N_{rays}. The y-axis denotes the number of timesteps within the corresponding bin.

oscillate, rather than tend smoothly toward equilibrium. Though not clear from the histograms, from $N_{\text{rays}} = 64$ to $N_{\text{rays}} = 16\,384$, the total number of steps is halved.

Test III

Test III provides an opportunity to examine the effect of other parameters on the runtime, including the dataset size. For the latter, recall from Section 5.3 that test3-128-smooth and test3-170-smooth are created very similarly, differing primarily in their total particle counts and smoothing radii. The wall-lock time as a function of simulation time for various configurations is given in Figure 5.48. Again, 10^8 total rays is a reasonable number for SPHRAY, achieving convergence.

For the $N = 128^3$ datasets, mid- and light-coloured lines in Figure 5.48, TARANIS runtimes are a factor of a few better than that of SPHRAY. Again, this is not particularly impressive relative to what might be achieved with a parallel SPHRAY code. Interestingly, SPHRAY is essentially unaffected by the smoothness of the underlying particle distribution, while slightly shorter runtimes are seen for TARANIS using the smoother field. Figure 5.49 shows that this is due to a slight decrease in the timestep size for the smoother initial condition (compare purple and light-purple lines). The underlying cause is, however, not clear.

We again see the unintuitive decreased runtimes for increased N_{rays}, being quite

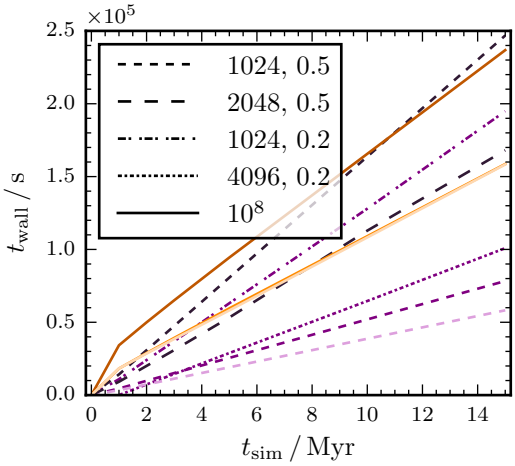


Figure 5.48. CRTCP test III wall-clock time as a function of simulation time for TARANIS (purples) and SPHRAY (oranges). For TARANIS, values in the legend denote the number of rays per timestep and the timestep multiplier; for SPHRAY, the total number of rays. Light purple and light orange lines are for the test3-128 dataset. Purple and orange lines are for the test3-128-smooth dataset (which was used in Section 5.3). Dark orange and dark purple lines are for the test3-170-smooth dataset.

dramatic for $f = 0.2$. The explanation is nonetheless as before: a combination of an implementation inefficiency of TARANIS, non-trivial scaling in N_{rays} for GRACE, and increased timestep sizes under better sampling, as per Figure 5.49.

Comparing now to the $N = 170^3$ dataset, dark purple lines in Figure 5.48, scaling in the number of particles is particularly poor for the $N_{\text{rays}} = 1024$ and $f = 0.5$ runs. Naïvely we expect the ray-tracing process to scale as approximately $N_{\text{rays}} N^{1/3} \log N$, here giving a factor of 1.4 at constant N_{rays}. However, it was shown in Chapter 4, Section 9.5 that the packet traversal method of GRACE becomes less effective at constant N_{rays} as the number of particles is increased, hence we actually observe worse scaling. Runtime has increased by a factor of ~ 3. Changes to the timestep account for a factor of ~ 2, with the distributions shown in Figure 5.49. The additional factor of ~ 1.5 is due to GRACE. The situation is markedly improved for $N_{\text{rays}} = 2048$, where we also see similarities in the timestep distributions (compare top- and bottom-left plots of Figure 5.49). This is expected, as $N_{\text{rays}} = 2048$ at $N = 170^3$ results in approximately equivalent (somewhat better) sampling to $N_{\text{rays}} = 1024$ at $N = 128^3$. The total number of timesteps in the former case is actually $\sim 5\%$ lower (not shown).

Fractional contributions to the total runtime present a similar picture as for test II. The tracing and rate-computation split is similar to Table 5.7, and suggestions made previously to reduce the tracing contribution apply here.

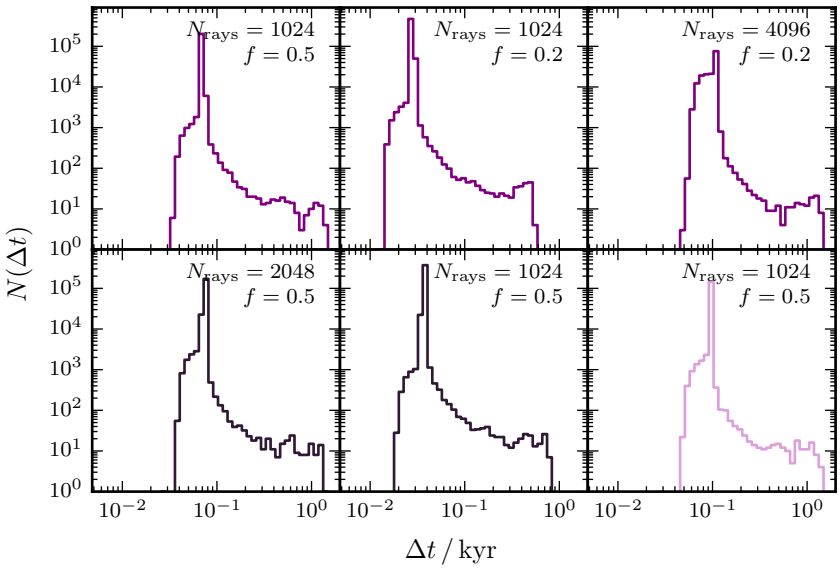


Figure 5.49. CRTCP test III histograms of timestep size for various values of $N_{\rm rays}$ and f. The y-axis denotes the number of timesteps within the corresponding bin, and colour-coding is as in Figure 5.48: light purple lines for the test3-128 dataset; purple lines for the test3-128-smooth dataset; and dark purple lines for the test3-170-smooth dataset.

In summary, performance here is good relative to the serial SPHRAY code, but is not expected to compare favourably to a parallel implementation. Scaling in the number of particles is also worse than expected for constant $N_{\rm rays}$, but as expected where the number of rays is increased to maintain constant sampling resolution.

Test IV

It has already been stated that TARANIS performs particularly poorly for test IV. In the interests of completeness, TARANIS and SPHRAY runtimes for both the (noisy) 128^3 and (smoother) 64^3 datasets are given in Figure 5.50; timestep distributions are given in Figure 5.51. Other parameters are as in Section 5.4; recall that, for the test4-128 dataset, a minimum timestep was enforced. TARANIS is over an order of magnitude slower than SPHRAY in both cases.

The test4-64-smooth timestep distribution suggests that, with a higher ray count and better sampling, the runtime for test4-128 will be reduced, though the smoothness of the smaller dataset may also be a factor. The point is moot, however, given that the test4-64-smooth runtime is still substantially slower than that of SPHRAY.

Fractional contributions show a 60–23 percentage split for tracing and rate-computation, respectively, for the 128^3 dataset. Since both processes occur in a loop

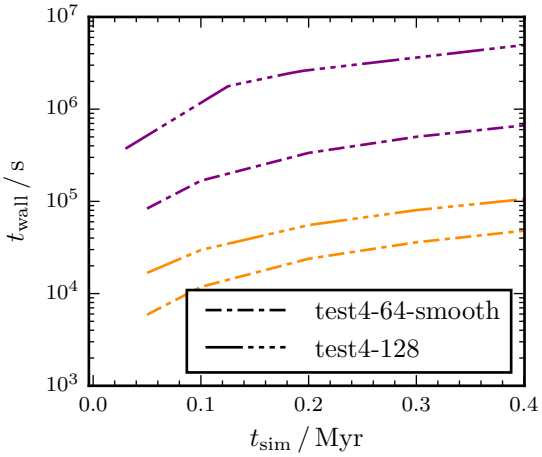


Figure 5.50. CRTCP test IV wall-clock time as a function of simulation time for TARANIS (purple) and SPHRAY (orange) for several initial condition datasets.

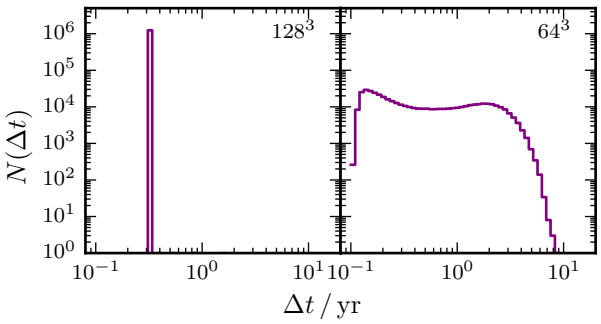


Figure 5.51. CRTCP test IV histograms of timestep size for several initial condition datasets.

over all 16 sources, it is expected that their total will be higher than for other tests. However, their relative values have also changed; here ~ 2.6, but ~ 1.4 for equal N and N_{rays} in test II. In part, this is because, as noted in Chapter 4, Section 9.5, at low ray counts sources near the simulation boundaries will further increase GRACE traversal times. A factor of ~ 1.2 can also be attributed to the fact that, regardless of source location, GRACE traces the test IV data set slower than it traces a glass dataset. As a final possible cause, note that TARANIS may compute photoionization and heating rates faster for very large and very small optical depths, avoiding a costly exponential.

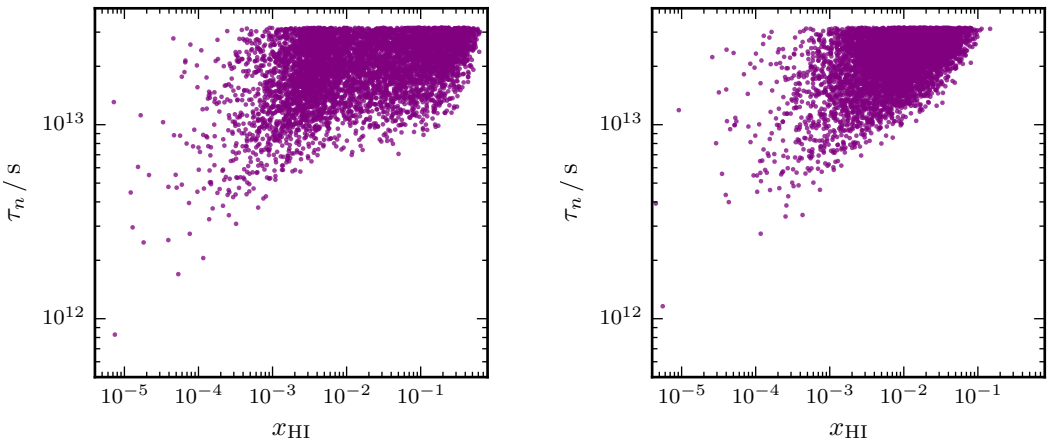
For $N_{\text{rays}} = 1024$, rays from all 16 sources could be traced in a single call to GRACE, reducing the tracing contribution by a factor of ~ 5. Ray tracing would then execute in approximately half the time of rate computation, and the total run time would also be approximately halved.

5.7 Routes to improved performance

In Section 5.6, it was shown that more-efficient use of GRACE would result in the rate computation stage dominating the runtime for $N_{\text{rays}} \gtrsim 1024$ at $N = 128^3$. Now, the poor scaling of GRACE in terms of N, for fixed N_{rays} per source, means rate computation will not remain dominant. However, for correct sampling (or merely to maintain a constant timestep size), we actually require $N_{\text{rays}} \propto N^{2/3}$, which conveniently maintains packet traversal effectiveness in GRACE, and hence expected scaling.

Nonetheless, rate computation then scales only as N, while ray traversal scales as $N \log N$; at some point, ray tracing will inevitably become dominant. To estimate the value of N at which this occurs, recall from Section 5.6 that with 16 sources and $N_{\text{rays}} = 1024$ per source in a 128^3 dataset, efficient use of GRACE will result in rate computation time being approximately double that of ray tracing. The above scaling relations then predict that the two will become equal when $\log(N) \,/\, \log(128^3) \approx 2$, or $N \approx 4 \times 10^{12}$. This is well beyond the number of particles which may be processed on a single GPU, and will remain so for the foreseeable future.

It therefore serves here to consider only improvements to the non-tracing components of TARANIS. Unfortunately, rate computation is in principle a straightforward operation, and no significant optimizations are obvious. Certainly, profiling the application is expected to present some avenues for improvement, but a method to increase the minimum timestep size is essential, particularly given timesteps of $< 1\,\text{yr}$ in test IV,



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 500\,\mathrm{Myr}$.

Figure 5.52. Test II neutral fraction against ionization timescale, τ_n, (in seconds) for all particles with $\tau_n < 10^{13.5}\,\mathrm{s}$.

and the fact that good convergence is reached with substantially lower-than-expected ray counts.

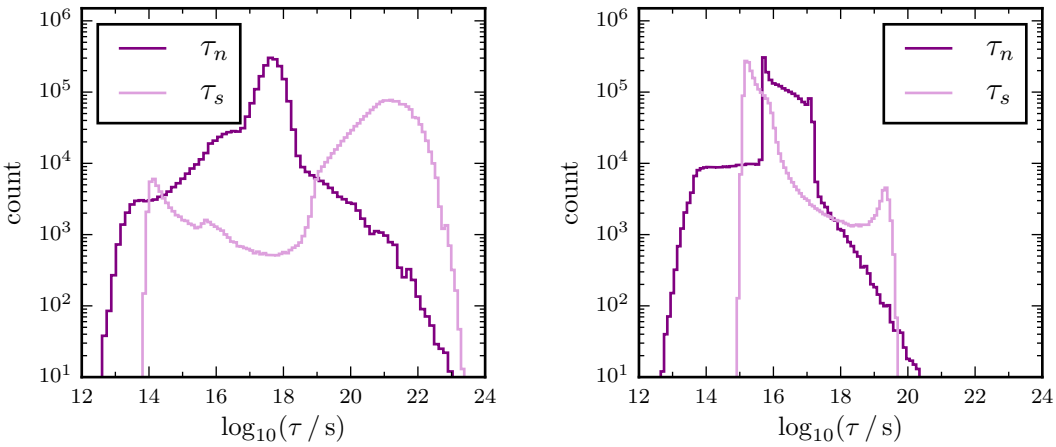
To that end, the ionization and entropy timescales (recall Section 4.5) are computed for TARANIS snapshots (which by default include all required particle state information, including ionization rates). Indeed, it is found that the globally-shortest timescale is always the ionization timescale, τ_n. Specifically, the minimum timestep is always due to a particle with a very low neutral fraction, but which is nonetheless limited by its ionization rate. This relationship is shown for test II (1024 rays) at early and late times in Figure 5.52.

These values of τ_n are put into context by examining the histograms of the limiting ionization or recombination (as appropriate) timescales, τ_n, and entropy-limited timescales, τ_s, for all particles, shown in Figure 5.53.

Similar results are seen for test III, though there the limiting particle typically has a larger neutral fraction, in the region of a few times 10^{-3}, and the total range of timescales spans 10^{10}–10^{22}. Test IV again shows similar behaviour, though more severe: particles with $x_{\mathrm{HI}} \lesssim 10^{-4}$ have minimum timescales which are two orders of magnitude smaller than the minimum timescale of particles with $x_{\mathrm{HI}} \gtrsim 10^{-4}$.

Do note that a low value of x_{HI} is not a good predictor of small timestep — many such particles have large timesteps — but that the smallest-allowed timestep is always due to a highly-ionized particle limited by its ionization timescale.

It makes some intuitive sense that this would be the case: we must ensure a small



(a) $t = 10\,\mathrm{Myr}$. **(b)** $t = 500\,\mathrm{Myr}$.

Figure 5.53. Test II, histograms of ionization state, τ_n, and entropy state, τ_s, timescales (both in seconds) for all particles.

timestep when ionizing a particle which is close to fully-ionized, otherwise we risk computing a nonphysical value $x_{\mathrm{HI}} < 0$. However, we would also expect all or most particles to be near-equilibrium at medium-to-late times, and so it is surprising that ionization rates should so-exceed recombination rates. This behaviour may in part be a result of using an explicit integration scheme to solve a stiff equation.

A naïve method to improve performance would then be to ignore all timescales for $x_{\mathrm{HI}} < k$, where k is a tunable parameter. Special care then needs to be taken to set the neutral fraction to zero where it will, almost certainly, become negative in those particles for which the timestep was too large. This has consequences for the correctness of results, requiring convergence tests in k.

A better solution may be to effectively mask-off particles with low x_{HI} and low τ_n, and evolve them over the too-larger timestep using an implicit method, such as backward Euler (used in SPHRAY) or the method of Anninos et al. (1997). Such particles could in fact be solved on the CPU, provided their number was relatively few; the benefit is two-fold, since variable sub-cycle iteration counts are unlikely to result in efficient use of the GPU. Solving via an implicit method in a non-causal manner, i.e. not in distance-order along the ray, no longer guarantees photon conservation. However, given that a minority of particles are expected to be affected, and that low neutral fractions imply high transmission in any case, this may be tolerable.

Correctness aside, both of these suggestions will prove difficult to implement effectively. The former requires *a priori* knowledge of a correct value of k. For tests

II through IV, respectively, values of approximately 3×10^{-3}, 3×10^{-3} and 1×10^{-4} would eliminate the small number of very-small timescale particles, showing that a useful k may well be unpredictable. Occasional use of an implicit scheme requires, similarly, a definition of 'small τ_n'. A fast, coarse binning of timesteps, and elimination of the bottom $\sim 1\%$ would be effective here, but its generality is not clear. Interestingly, masking-off all particles with $\tau_n < 0.1\tau_s^{\min}$ is a particularly good definition across all tests, but without further testing or theoretical motivation is also dubious.

If no such generally-applicable solution can be found, it seems that moving to a fully-implicit solver will be necessary. When ensuring photon conservation, this brings with it scaling issues noted for SPHRAY and P-SPHRAY (recall Chapter 2, Section 4.3) and certainly will be challenging to implement efficiently on the GPU. To reiterate, the poor scaling is the result of two factors: the necessity to solve causally, from first-to-last intersection, in order to ensure photon conservation; and the fact that, while a ray-particle intersection is being processed, no other thread can update that particle. Close to the source in particular, or in a many-source environment, this leads to stalled threads, with no particle available for updating.

Mackey (2012), for example, have attempted to address the first of these concerns. They suggest a multi-step method: first, compute rates and update the system, in parallel, from t to $t + \Delta t/2$; second, recompute all rates using this intermediate state; finally, and again in parallel, perform a full step from t to $t + \Delta t$. This does not require rays to be processed in-order, and is stated to give second-order photon conservation. The potential gains come from the fact that this method retains accuracy when the timestep is proportional merely to the inverse of the rate $|dn_{\rm HI}/dt|$, not the fractional change (as used here), hence larger timesteps are possible. It does, however, require all ray tracing to be performed twice, or for the resulting particle integrals to be stored for all ray-particle intersections. Neither of these options is particularly desirable when dealing with multiple sources. Additionally, the state of all particles at both t and $t + \Delta t/2$ must be stored over a timestep. In practice, then, one is not necessarily hopeful that this offers a solution.

The only obvious direction remaining is to solve rays in-order but reduce the time taken to perform a step of the implicit solver. This will reduce the time over which a particle is locked and reduce contention, improving scaling. Of course, the scaling will remain effective only up to some limit, and the goal is to ensure this limit is large-enough.

To that end, stiffly-stable non-linear explicit methods might be employed. However, these typically require values of the derivative over several orders, estimates of which typically require the previous n derivatives to be stored to approximate the current nth-order derivative.

Finally, note that several other stages of the TARANIS implementation become comparable to tracing or rate computation when only one of N and N_{rays} are large. In particular, computing cumulative column densities (which is proportional to the number of intersections) and copying per-particle ionization fractions to the GPU (a prerequisite for column densities) are significant for large N_{rays} and large N, respectively. Ionization fractions must also be re-ordered such that they match the particle order used for tracing, i.e. the Morton-key order, introducing additional overhead. Both become significant in part due to the fact that TARANIS splits its ray tracing and particle state-evolution over two GPUs, where available. This necessitates GPU-host-GPU transfers, and currently the re-ordering is performed by the GPU. These edge-cases may be mitigated by forgoing the copy entirely when only one GPU is in use (as has been the case for all results presented here); performing the re-ordering on the GPU; and overlapping tracing with other stages when multiple GPUs are used, for example by tracing the next source's rays while simultaneously computing rates due to the current source.

6 Conclusion

In this chapter I have introduced a new radiative transfer code, TARANIS, which operates directly on a smoothed particle hydrodynamics (SPH) field and runs almost entirely on NVIDIA CUDA-capable GPUs. Its aim is to enable high-performance, yet accurate, ray-traced radiative transfer simulations for a larger body of researchers. The underlying algorithm was presented in Section 3; ionization and heating rates are accumulated for each particle, thus allowing particles to be updated completely in parallel, while an explicit Euler integration scheme ensures photon conservation. Other aspects of the implementation — which has not been published in its current form — were discussed in Section 4. The ray tracing, ray generation and cumulative column-density computations are all performed by GRACE, and these stages represent my main contribution to the TARANIS codebase.

In Section 5, the Cosmological Radiative Transfer Comparison Project (CRTCP) was introduced, a suite of standard test cases for cosmological radiative transfer codes. This is used to assess the accuracy of TARANIS, as well as its performance in Section 5.6. I have also run the CRTCP test suite with SPHRAY, an alternative radiative transfer code for SPH datasets, as a more direct means of comparison. Results for tests I through III are generally excellent, and where codes disagree TARANIS often closely matches at least one of RSPH, SPHRAY and the equilibrium-chemistry solver, RABACUS; in cases where there is more substantial disagreement, TARANIS is within the spread of values generated by the other codes. Performance on these tests is acceptable, and generally TARANIS out-performs SPHRAY. However, it is expected that a parallel SPHRAY code would outperform TARANIS, which is a cause for concern given the stated goals. Results for test IV also appear generally correct, but performance here is incredibly poor — one to two orders of magnitude slower than SPHRAY— which also prevented a thorough test of convergence. Particularly as this is by far the most realistic test of the four, being a simple cosmological reionization simulation, it must be concluded that TARANIS is not yet suitable for use by the wider research community, and in fact further effort — possibly substantial — is necessary before it can be deemed a success.

To that end, in Section 5.7 I have identified a limiting factor to performance, namely the extremely short timesteps which are typically required by only a very small fraction of mostly-ionized particles. Several potential solutions have been put forward to alleviate this problem, the investigation of which is the clear next-step in the evolution of this code.

Appendices

Appendix A

Hydrogen Ionizing Photon Flux of a Black Body

Planck's law for the *spectral radiance* (radiated energy per unit time per unit area per unit solid angle per unit frequency; *radiance* per unit frequency) of a black body is

$$B_\nu(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{e^{\frac{h\nu}{k_\mathrm{B}T}} - 1}, \tag{A.1}$$

where ν is frequency, T is (absolute) temperature, h is Planck's constant, c is the speed of light in a vacuum and k_B is the Boltzmann constant. In SI, the units of B_ν are $\mathrm{J\,s^{-1}\,m^{-2}\,sr^{-1}\,Hz^{-1}}$. Radiated energy can be converted to radiated photon count by dividing through by the energy of a photon at frequency ν, $h\nu$. Thus,

$$B_\gamma(\nu, T) = \frac{2\nu^2}{c^2} \frac{1}{e^{\frac{h\nu}{k_\mathrm{B}T}} - 1}, \tag{A.2}$$

where B_γ has units of photons $\mathrm{s^{-1}\,m^{-2}\,sr^{-1}\,Hz^{-1}}$.

Now, to compute the total photon flux, or photons per unit time per unit area, B_γ must be integrated over solid angle and frequency. For the latter, consider the hemisphere into which a surface element of the black body, $\mathrm{d}A$, radiates. A black body is a Lambertian radiator; that is, its spectral radiance is independent of direction. This is due to the isotropy of radiation in its interior. As such, Lambert's cosine law[1] must

[1] Consider a surface element, $\mathrm{d}A$, viewed from a direction making an angle θ with the normal to the surface at $\mathrm{d}A$, emitting into a solid angle $\mathrm{d}\Omega$. While B_γ itself is independent of θ, the solid angle into

be employed, and we integrate the quantity $B_\gamma \cos\theta \, \mathrm{d}\Omega = B_\gamma \cos\theta \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi$ over the hemisphere to obtain the photon *spectral exitance*,

$$M_{\gamma,\,\nu} = \int_0^{2\pi} \mathrm{d}\phi \int_0^{\frac{\pi}{2}} \mathrm{d}\theta \, B_\gamma \cos\theta \sin\theta = \pi B_\gamma .$$

Now, to obtain the photon *radiant exitance*, a final integration over frequency must be performed. In particular, the integral is over all frequencies greater than the ionization threshold frequency of hydrogen, ν_0,

$$M_\gamma = \pi \int_{\nu_0}^{\infty} \mathrm{d}\nu \, B_\gamma . \tag{A.3}$$

This integral is not so trivial. First, consider

$$\int \mathrm{d}\nu \, B_\gamma = \int \mathrm{d}\nu \, \frac{2\nu^2}{c^2} \frac{1}{e^{\frac{h\nu}{k_\mathrm{B}T}} - 1},$$

and make the substitution $x = {}^{h\nu}/_{k_\mathrm{B}T}$, resulting in

$$M_\gamma = \frac{2\pi}{c^2} \left(\frac{k_\mathrm{B}T}{h} \right)^3 \int_{x_0}^{\infty} \mathrm{d}x \, \frac{x^2}{e^x - 1} \equiv \alpha \, \mathcal{J}\Big|_{x_0}^{\infty}, \tag{A.4}$$

where $x_0 \equiv {}^{h\nu_0}/_{k_\mathrm{B}T}$.

Now, note that,

$$\mathcal{J} = \int \mathrm{d}x \, x^2 \frac{e^{-x}}{1 - e^{-x}},$$

and,

$$\frac{1}{1-y} = \sum_{n=0}^{\infty} y^n \text{ for } |y| < 1,$$

so, letting $y = e^{-x}$,

$$\mathcal{J} = \int \mathrm{d}x \, x^2 e^{-x} \sum_{n=0}^{\infty} e^{-nx} = \int \mathrm{d}x \, x^2 \sum_{n=0}^{\infty} e^{-(n+1)x} = \sum_{n=1}^{\infty} \int \mathrm{d}x \, x^2 e^{-nx}, \tag{A.5}$$

where one should note the change in the starting value for the summation in the final step. Integration by parts may be employed to solve the integral in Eq. (A.5), giving

$$\mathcal{J}\Big|_{x_0}^{\infty} = -\sum_{n=1}^{\infty} \left[\frac{(nx)^2 + 2nx + 2}{n^3} \, e^{-nx} \right]_{x_0}^{\infty}, \tag{A.6}$$

which the surface element emits is not. It is modulated precisely as $\cos\theta$. Put slightly differently, the apparent size, or angular size, of the surface element $\mathrm{d}A$ is directly proportional to $\cos\theta$.

where the limits on $\mathcal{J}$ have been reintroduced. The upper limit is zero: let $y = nx$, then

$$\lim_{y\to\infty}(y^2 + 2y + 2)e^{-y} = \lim_{y\to\infty} 2e^{-y} = 0,$$

where the first equivalence follows from L'Hôpital's rule. Hence,

$$\mathcal{J}\Big|_{x_0}^{\infty} = \sum_{n=1}^{\infty}\left(x_0^2\,\frac{e^{-nx_0}}{n} + 2x_0\,\frac{e^{-nx_0}}{n^2} + 2\,\frac{e^{-nx_0}}{n^3}\right). \tag{A.7}$$

Relevant now is the polylogarithm, defined as

$$\mathrm{Li}_s(z) = \sum_{n=1}^{\infty}\frac{z^n}{n^s}, \tag{A.8}$$

and valid for arbitrary complex order s and all complex arguments $|z| < 1$. For the special case $s = 1$, $\mathrm{Li}_1(z) = -\ln(1 - z)$. Setting $z = e^{-x_0}$ one finds

$$\mathcal{J}\Big|_{x_0}^{\infty} = 2x_0\,\mathrm{Li}_2(e^{-x_0}) + 2\,\mathrm{Li}_3(e^{-x_0}) - x_0^2\,\ln(1 - e^{-x_0}).$$

Substituting into Eq. (A.4),

$$M_\gamma(T) = \frac{2\pi}{c^2}\left(\frac{k_\mathrm{B}T}{h}\right)^3\left[2x_0\,\mathrm{Li}_2(e^{-x_0}) + 2\,\mathrm{Li}_3(e^{-x_0}) - x_0^2\,\ln(1 - e^{-x_0})\right] \tag{A.9}$$

gives the total number of ionizing photons per unit time per unit area emitted by a black body at absolute temperature T.

Numerically,[2] $M_\gamma(T = 10^5\,\mathrm{K}) = 1.0677 \times 10^{30}\,\mathrm{s^{-1}\,m^{-2}}$. The surface area A of a $T = 10^5\,\mathrm{K}$ black body emitting $\dot{N}_\gamma$ hydrogen-ionizing photons per second is therefore,

$$A = \frac{\dot{N}_\gamma}{1.0677 \times 10^{30}}\,\mathrm{m^2}. \tag{A.10}$$

[2] Computed using the `mpmath` Python package, available at `http://mpmath.org`. Assumed numerical values of $c = 299\,792\,458\,\mathrm{m\,s^{-1}}$, $k_\mathrm{B} = 1.380\,65 \times 10^{-23}\,\mathrm{J\,K^{-1}}$, $h = 6.626\,07 \times 10^{-34}\,\mathrm{J\,s}$, $\nu_0 = {}^{\Phi_0}/_{h}$ where $\Phi_0 = 13.5984\,\mathrm{eV}$ and $1\,\mathrm{eV} = 1.602\,18 \times 10^{-19}\,\mathrm{J}$.

Appendix B

Generating Uniform Random Directions

1 Uniform Directions Within a Solid Angle

1.1 Distributions Centred on the z-axis

Consider a conical section of a sphere, with an apex angle of α and, thus, subtending a solid angle of $\Omega = 2\pi(1 - \cos\frac{\alpha}{2})$. The spherical cap is the area on the surface of the sphere bounded by this cone. Here, a uniform distribution within a solid angle is defined such that it generates points uniformly on this spherical cap (that is, uniformly on the surface of the sphere, rather than uniformly on the base of the spherical cap).

For a given solid angle, ω, the probability density function (PDF) must be constant for all such values of ω, independent of direction. Thus,

$$p(\omega)\,\mathrm{d}\omega = c\,\mathrm{d}\omega, \tag{B.1}$$

where c is a constant and $p(\omega)$ is the PDF. The integral of this quantity over the entire solid angle of interest, Ω (that is, the solid angle within which we wish to generate random direction vectors) must be equal to one;

$$\int_{\Omega} p(\omega)\,\mathrm{d}\omega = 1,$$

and one obtains the intuitive result

$$c = \frac{1}{\Omega}. \tag{B.2}$$

Further, in spherical polar coordinates, we have

$$\mathrm{d}\omega = \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi, \tag{B.3}$$

where the polar angle is $\theta \in [0, \pi)$ and the azimuth angle is $\phi \in [0, 2\pi)$.

Now, for some continuous random variable X with PDF $p_X(x)$, consider the variable $Y = g(X)$ with PDF $p_Y(y)$ and $y = g(x)$. It holds that

$$|p_Y(y) \, \mathrm{d}y| = |p_X(x) \, \mathrm{d}x|,$$

so,

$$p(\omega) \, \mathrm{d}\omega = p(\theta, \phi) \, \mathrm{d}\theta \, \mathrm{d}\phi.$$

Substituting Eqs (B.2) and (B.3) into the above gives

$$p(\theta, \phi) = \frac{\sin\theta}{\Omega}. \tag{B.4}$$

We may marginalize over the random variable Φ to obtain the marginal probability density function of Θ, $p_\Theta(\theta)$; that is, the probability density of Θ when Φ is not known. This is

$$p_\Theta(\theta) = \int_0^{2\pi} p(\theta, \phi) \, \mathrm{d}\phi = \frac{2\pi \sin\theta}{\Omega}. \tag{B.5}$$

The conditional density function $p_\Phi(\phi \mid \theta)$, that is, the distribution of Φ for a given value of Θ, can now be computed as,

$$p_\Phi(\phi \mid \theta) \equiv \frac{p(\theta, \phi)}{p_\Theta(\theta)} = \frac{1}{2\pi}, \tag{B.6}$$

where one will note that there is no θ dependence. This is as expected: for a given polar angle, the distribution of directions around the azimuth angle is uniform. (The first equivalence above is clear if one multiplies through by $p_\Theta(\theta)$.)

Direction vectors satisfying the desired uniform distribution within Ω may be generated by first producing values of Θ which satisfy Eq. (B.5), and then values of Φ

which satisfy Eq. (B.6). (Actually, because Eq. (B.6) is a constant, the variables may be generated in any order.) One such method to achieve this is inversion sampling. First, we compute the cumulative distribution functions (CDFs) for Θ and Φ,

$$F_\Theta(\theta) = \int_0^\theta p_\Theta(\theta')\,\mathrm{d}\theta' = \frac{2\pi}{\Omega}(1-\cos\theta), \tag{B.7a}$$

$$F_\Phi(\phi) = \int_0^\phi p_\Phi(\phi' \mid \theta)\,\mathrm{d}\phi' = \frac{1}{2\pi}\phi. \tag{B.7b}$$

Then, two independent uniform random variables u and v, both $\in [0, 1)$, are chosen (which may be generated by any suitable software library). Setting $u = F_\Theta(\theta)$ and $v = F_\Phi(\phi)$ gives

$$\theta = \arccos\left(1 - \frac{\Omega u}{2\pi}\right), \tag{B.8a}$$

$$\phi = 2\pi v. \tag{B.8b}$$

Typically, a normalized direction vector specified in Cartesian co-ordinates is desired. Taking the spherical polar co-ordinate conversions $x = \sin\theta\cos\phi$, $y = \sin\theta\sin\phi$ and $z = \cos\theta = \sqrt{1-x^2-y^2}$, where we have assumed a unit sphere, and recalling the identity $\sin^2\theta = 1 - \cos^2\theta$, one finds

$$x = \sqrt{1-z^2}\cos(2\pi v), \tag{B.9a}$$

$$y = \sqrt{1-z^2}\sin(2\pi v), \tag{B.9b}$$

$$z = 1 - \frac{\Omega u}{2\pi}. \tag{B.9c}$$

It is then clear that we may instead generate two random variables $\beta \equiv 2\pi v \in [0, 2\pi)$ and $z \in (1 - \Omega/2\pi,\ 1]$. From the definition of Ω in terms of the apex (opening) angle of a cone, α, we have $\Omega = 2\pi(1 - \cos\frac{\alpha}{2})$, and it follows that $z \in [\cos\frac{\alpha}{2},\ 1)$. Finally, then,

$$x = \sqrt{1-z^2}\cos(\beta), \tag{B.10a}$$

$$y = \sqrt{1-z^2}\sin(\beta), \tag{B.10b}$$

$$z \in \mathcal{U}\left(\cos\frac{\alpha}{2},\ 1\right), \tag{B.10c}$$

where $\beta \in \mathcal{U}(0,\ 2\pi)$ and $\mathcal{U}(a,\ b)$ represents a continuous uniform distribution between a and b.

Choosing any two independent z and β as described above and inserting them into Eqs (B.10) produces the Cartesian co-ordinates of a normalized direction vector sampled from a distribution which is uniform within a cone of opening angle α centred on the positive z-axis. Note that this holds for any value of $\alpha \in [0, 2\pi]$, where values $\geq \pi$ no longer correspond directly to cones and are better thought of in terms of the resulting solid angle Ω. A value of $\alpha = 2\pi$ corresponds to generating uniform direction vectors over the entire surface of the unit sphere.

1.2 Distributions Centred on an Arbitrary Axis

Section 1.1 generates a distribution centred on the positive z-axis, henceforth defined as $\hat{k}$, where the circumflex denotes a unit vector. Typically, having such a distribution centred around an arbitrary vector $\hat{d}$ is desired. This can be achieved by computing the transformation matrix U such that $\hat{d} = U\hat{k}$, and applying this matrix to all generated vectors.

To compute U, first consider the convenient basis,

$$\hat{u} = \hat{k}, \tag{B.11a}$$

$$\hat{v} = \frac{\hat{d} - (\hat{k} \cdot \hat{d})}{|\hat{d} - (\hat{k} \cdot \hat{d})|}, \tag{B.11b}$$

$$\hat{w} = \hat{u} \times \hat{v}. \tag{B.11c}$$

In this basis, the rotation from $\hat{k}$ to $\hat{d}$ about $\hat{w}$ is described by the rotation matrix

$$R_w(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{B.12}$$

where θ is the angle between $\hat{k}$ and $\hat{d}$ as measured in the plane containing both $\hat{k}$ and $\hat{d}$, and in the clockwise sense when looking in direction $\hat{w}$. We therefore also have the identities,

$$\hat{k} \cdot \hat{d} = \cos\theta, \tag{B.13a}$$

$$\hat{k} \times \hat{d} = \sin\theta\, \hat{w}. \tag{B.13b}$$

Defining the orthogonal basis matrix M as,

$$M = \begin{pmatrix} \hat{u}_1 & \hat{v}_1 & \hat{w}_1 \\ \hat{u}_2 & \hat{v}_2 & \hat{w}_2 \\ \hat{u}_3 & \hat{v}_3 & \hat{w}_3 \end{pmatrix}, \tag{B.14}$$

the rotation Eq. (B.12) may be described in the standard basis as,

$$U = M R_w(\theta) M^{-1}. \tag{B.15}$$

In full (and noting $M^{-1} = M^{\mathsf{T}}$),

$$U = \begin{pmatrix} \hat{k}_1 & \dfrac{\hat{d}_1 - c\hat{k}_1}{s} & \hat{w}_1 \\ \hat{k}_2 & \dfrac{\hat{d}_2 - c\hat{k}_2}{s} & \hat{w}_2 \\ \hat{k}_3 & \dfrac{\hat{d}_3 - c\hat{k}_3}{s} & \hat{w}_3 \end{pmatrix} \begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \hat{k}_1 & \hat{k}_2 & \hat{k}_3 \\ \hat{v}_1 & \hat{v}_2 & \hat{v}_3 \\ \hat{w}_1 & \hat{w}_2 & \hat{w}_3, \end{pmatrix}$$

where $s \equiv \sin\theta$ and $c \equiv \cos\theta$, and in the left matrix Eqs (B.11) have been used. Expanding, one finds

$$\begin{aligned} U_{ij} &= \hat{d}_i\hat{k}_j - s\hat{k}_i\hat{v}_j + c\hat{v}_i\hat{v}_j + \hat{w}_i\hat{w}_j \\ &= \hat{d}_i\hat{k}_j - \hat{d}_j\hat{k}_i + c(\hat{k}_i\hat{k}_j + \hat{v}_i\hat{v}_j) + \hat{w}_i\hat{w}_j, \end{aligned}$$

where the second equality simply applies Eq. (B.11b). Adding the zero-valued term $c(\hat{w}_i\hat{w}_j - \hat{w}_i\hat{w}_j)$ to both sides and noting that, due to Eq. (B.13b), $\hat{d}_i\hat{k}_j - \hat{d}_j\hat{k}_i = -s\epsilon_{ijk}w_k$, we achieve

$$U_{ij} = -s\epsilon_{ijk}\hat{w}_k + c(\hat{k}_i\hat{k}_j + \hat{v}_i\hat{v}_j + \hat{w}_i\hat{w}_j) + (1-c)\hat{w}_i\hat{w}_j, \tag{B.16}$$

To (somewhat) simplify this expression, first consider the diagonal terms $i = j$. Hence, $\epsilon_{ijk} = 0$ by definition, and $\hat{k}_i\hat{k}_j + \hat{v}_i\hat{v}_j + \hat{w}_i\hat{w}_j = \delta_{ij} = 1$ follows from the properties of the orthogonal matrix M. For terms not on the diagonal, $i \neq j$ and M instead leads to $\hat{k}_i\hat{k}_j + \hat{v}_i\hat{v}_j + \hat{w}_i\hat{w}_j = \delta_{ij} = 0$. Finally, recall $\hat{w}_i = (\hat{k} \times \hat{d})_i \,/\, s$. Thus,

$$U = \left[\hat{k} \times \hat{d}\right]_\times + c\mathbb{1} + \frac{1-c}{s^2}\left[(\hat{k} \times \hat{d}) \otimes (\hat{k} \times \hat{d})\right],$$

where $[\,]_\times$ denotes the skew-symmetric cross product matrix

$$[\mathbf{a}]_\times \equiv \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix},$$

defined such that $[\mathbf{a}]_\times \mathbf{b} \equiv \mathbf{a} \times \mathbf{b}$, and $\otimes$ is simply the matrix product, i.e. $\mathbf{a} \otimes \mathbf{b} \equiv \mathbf{a}\mathbf{b}^\mathsf{T}$. Using the identity $\mathbf{a} \otimes \mathbf{a} = [\mathbf{a}]_\times^2 + |\mathbf{a}|^2\,\mathbb{1}$ and that $|\hat{k} \times \hat{d}| = s$ [c.f. Eq. (B.13b)], one finally arrives at

$$U = \mathbb{1} + \left[\hat{k} \times \hat{d}\right]_\times + \frac{1-c}{s^2}\left[\hat{k} \times \hat{d}\right]_\times^2. \tag{B.17}$$

Now, for the relevant special case $\hat{k} = (0,\ 0,\ 1)$ we have $\hat{k} \times \hat{d} = (-d_2,\ d_1,\ 0)$, $s^2 \equiv \sin^2\theta = 1 - d_3^2 = (1-d_3)(1+d_3)$ and $c \equiv \cos\theta = d_3$. After some cancelling,

$$U = \begin{pmatrix} 1 - \dfrac{d_1^2}{1+d_3} & \dfrac{-d_1 d_2}{1+d_3} & d_1 \\ \dfrac{-d_1 d_2}{1+d_3} & 1 - \dfrac{d_2^2}{1+d_3} & d_2 \\ -d_1 & -d_2 & d_3. \end{pmatrix} \tag{B.18}$$

The full procedure is then to generate a distribution following Eqs (B.10), then apply $\mathbf{p}' = U\mathbf{p}$ to each point $\mathbf{p}$ in said distribution, re-centring it about the desired direction $\hat{d}$.

However, Eq. (B.18) does not give well-defined values when $d_3 = -1$, i.e. $\theta = \pi$. One would expect to encounter numerical errors for values of θ which are close to π. To counter this, if $\hat{k} \cdot \hat{d} < 0$, i.e. $\theta > \pi/2$, we may instead generate points around $\tilde{k} \equiv -\hat{k}$. The necessary modification to Eqs (B.10) affects only the z component,

$$z \in \mathcal{U}\left(-1, -\cos\frac{\alpha}{2}\right). \tag{B.19}$$

Substituting $\tilde{k}$ for $-\hat{k}$ in Eq. (B.17) gives the equivalent of Eq. (B.18),

$$\tilde{U} = \begin{pmatrix} 1 - \dfrac{d_1^2}{1-d_3} & \dfrac{-d_1 d_2}{1-d_3} & -d_1 \\ \dfrac{-d_1 d_2}{1-d_3} & 1 - \dfrac{d_2^2}{1-d_3} & -d_2 \\ d_1 & d_2 & -d_3, \end{pmatrix} \tag{B.20}$$

which is well-defined over its domain $\pi/2 < \theta \leq 3\pi/4$ as $d_3 \leq 0$.

If $d_3 \geq 0$, one generates points via Eqs (B.10) and rotates with U in Eq. (B.18); if $d_3 < 0$, one generates points via Eq. (B.19) and uses $\tilde{U}$ in Eq. (B.20). The full procedure remains otherwise unaltered.

2 Uniform Directions On the Unit Sphere

An isotropic distribution of directions, i.e. a distribution which is uniform on the surface of the unit sphere, may be generated by the method described in Section 1. One simply takes the whole solid angle $\Omega = 4\pi$, hence $\alpha = 2\pi$ in Eqs (B.10), and some arbitrary value of $\hat{d}$. However, this is a little cumbersome, and for the special case of $\Omega = 4\pi$ there exists a (practically speaking) simpler method, described below.

Choose three independent standard normal random variables x, y and z; that is, three independent variables from the normal distribution $\mathcal{N}(0, 1)$ with mean $\mu = 0$ and variance $\sigma^2 = 1$. The probability density function is

$$f(x \mid \mu,\ \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}.$$

Then, the point

$$\hat{d} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} (x,\ y,\ z)$$

is uniformly distributed on the surface of the unit sphere. In fact, this technique applies to a hypersphere of any integer n dimensions.

To understand this in three dimensions, consider the probability density function of the vector $\vec{r} = (x,\ y,\ z)$,

$$\begin{aligned} f(\vec{r}) &= \left(\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}\right)\left(\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2}\right)\left(\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2}\right) \\ &= \frac{1}{(2\pi)^{\frac{3}{2}}} e^{-\frac{1}{2}r^2} \equiv f(r), \end{aligned}$$

where $r^2 \equiv x^2 + y^2 + z^2$. Clearly, this is invariant under rotations, depending only on the length of the vector. Since the mapping from $\vec{r}$ to $\hat{d}$ only modifies the length of the vector, the PDF $f(\hat{d})$ is similarly invariant under rotations, and hence uniform on the surface of the unit sphere.

Appendix C

Block-wide exclusive scan sum of booleans

A scan-sum of per-thread boolean values represents a special case, because there exist instructions for efficient warp-wide scan-sums of boolean values. Specifically `__ballot(predicate)` and `__popc(value)`. For the former, referring to the CUDA 8.0 programming guide,[1]

> *"Evaluate predicate for all active threads of the warp and return an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active."*

The `__popc(value)` instruction returns the number of bits set to 1 in `value`. If the exclusive scan is defined to be the number of lanes with lane IDs less than the current lane ID and which are holding a true predicate value, it may be implemented as in Listing C.1

Listing C.1. Warp-wide exclusive scan sum.

```
unsigned int bitfield = __ballot(predicate);
int total = __popc(bitfield);
int offset = __popc(bitfield & ((1 << lane)) - 1)
```

The above values are combined with a standard shared memory scheme for scan-summing per-warp values over a block

[1] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#warp-vote-functions`

Listing C.2. Block-wide exclusive scan sum.

```
// One thread from each warp writes to shared memory.
if (lane == 0) shared[wid] = total;
__syncthreads();

// One thread in the block performs a scan-sum of those values.
if (wid == 0) {
    int value = (lane < num_warps) ? shared[lane] : 0;

    // Inclusive scan.
    for (int i = 1; i < WARP_SIZE; i *= 2) {
        int reduction = __shfl_up(value, i);
        if (lane >= i) value += reduction;
    }
    shared[lane] = value;
}
__syncthreads();

offset += (wid > 0) ? shared[wid - 1] : 0;
```

Note that the `__shfl()` instructions are only available on Kepler and newer architectures. On Fermi, the inclusive scan could instead be achieved efficiently with a Hillis-Steele scan (Hillis and Steele Jr., 1986). This performs slightly worse, and requires double the shared memory.

Appendix D

Equivalence and noninferiority tests

1 Equivalence test for normal data

The two one-sided test (TOST) procedure for testing the equivalence of two normally-distributed samples is outlined here; for an accessible overview, within the context of bioequivalence testing, see for example Walker and Nowacki (2011), or Schuirmann (1987) for a more technical discussion.

Consider an observed and a reference sample with, respectively, sample sizes n_T and n_R; population means μ_T and μ_R; sample means η_T and η_R; and unbiased sample variances s_T^2 and s_R^2. To test for equivalence, first both a significance level, α, and a region of equivalence, $(-e_l, e_u)$, where $e_l, e_u > 0$, must be chosen. The region of equivalence specifies the largest deviation of the test sample mean from the reference mean which is deemed *practically* equivalent. It should be chosen appropriately for the situation at hand. The null hypothesis is

$$H : (\mu_T - \mu_R) \leq -e_l \quad \text{or} \quad (\mu_T - \mu_R) \geq e_u, \tag{D.1}$$

against the alternative

$$K : -e_l < (\mu_T - \mu_R) < e_u. \tag{D.2}$$

The confidence interval, (C_l, C_u), for the difference in the population means is computed

via its estimator, $\eta_T - \eta_R$, as (Berger and Hsu, 1996)

$$C_l = \min\left[0,\, (\eta_T - \eta_R) - S \cdot t_{1-\alpha,\,\nu}\right], \tag{D.3}$$

$$C_u = \max\left[0,\, (\eta_T - \eta_R) + S \cdot t_{1-\alpha,\,\nu}\right], \tag{D.4}$$

though note that unlike Berger and Hsu (1996), here S uses the more conservative Welch-Satterthwaite estimate, which does not assume equal variances in the observed and reference samples,

$$S = \sqrt{\frac{s_T^2}{n_T} + \frac{s_R^2}{n_R}}. \tag{D.5}$$

The (effective) degrees of freedom is then given by the Welch-Satterthwaite equation,

$$\nu = \frac{\left(\dfrac{s_T^2}{n_T} + \dfrac{s_R^2}{n_R}\right)^2}{\dfrac{s_T^4}{n_T^2(n_T-1)} + \dfrac{s_R^4}{n_R^2(n_R-1)}}, \tag{D.6}$$

and $t_{1-\alpha,\,\nu}$ is the $1-\alpha$ quantile of the Student's t-distribution with ν degrees of freedom. If the interval $(C_l,\, C_u)$ lies entirely within $(-e_l,\, e_u)$, the null hypothesis that the two distributions are different is rejected at a confidence level $1-\alpha$, with the alternative that $\eta_T - \eta_R$ lies within the region of equivalence.

2 Noninferiority test for non-normal data

The TOST procedure in the previous section assumes normally-distributed samples. When this is not (at least approximately) the case, an alternative is required. A non-parametric test for noninferiority (and equivalence) was presented by Wellek (1996), though also see Wellek (2010, Chapter 6), and the key results are reproduced below.

Again, consider an observed test distribution, T, and a reference distribution, R. We now concern ourselves with the functional $\gamma = P[T_i > R_j]$. Note that, if T and R are equivalent, we expect $P[T_i > R_j] = P[R_j > T_i] = 1/2$. Inferiority may correspond to smaller or larger values depending on the context; the two null hypotheses are, respectively,

$$H_1 : \gamma \leq \frac{1}{2} - e, \tag{D.7}$$

$$H_2 : \gamma \geq \frac{1}{2} + e. \tag{D.8}$$

The alternative hypotheses are

$$K_1 : \gamma > \frac{1}{2} - e, \tag{D.9}$$

$$K_2 : \gamma < \frac{1}{2} + e. \tag{D.10}$$

Here e quantifies the magnitude of the acceptable deviation of γ from $\frac{1}{2}$.

The estimator for γ is

$$\hat{\gamma} = \frac{1}{n_T\, n_R} \sum_{i=1}^{n_T} \sum_{j=1}^{n_R} I^+(T_i - R_j), \tag{D.11}$$

where $I^+(x)$ is 1 if $x > 0$ and 0 otherwise. The variance of $\hat{\gamma}$ is estimated as

$$\hat{\sigma}^2_{\hat{\gamma}} = \frac{1}{n_T\, n_R} \left[\hat{\gamma} - (n_T + n_R - 1)\hat{\gamma}^2 + (n_T - 1)\hat{\gamma}_{TTR} + (n_R - 1)\hat{\gamma}_{TRR}\right], \tag{D.12}$$

where $\hat{\gamma}_{TTR}$ and $\hat{\gamma}_{TRR}$ are themselves estimators for

$$\gamma_{TTR} = P[T_{i_1} > R_j,\, T_{i_2} > R_j], \tag{D.13}$$

$$\gamma_{TRR} = P[T_i > R_{j_1},\, T_i > R_{j_2}], \tag{D.14}$$

where $i_2 > i_1$ and $j_2 > j_1$. They are defined as

$$\hat{\gamma}_{TTR} = \frac{2}{n_T(n_T - 1)n_R} \sum_{i_1=1}^{n_T-1} \sum_{i_2=i_1+1}^{n_T} \sum_{j=1}^{n_R} I^+(T_{i_1} - R_j) \cdot I^+(T_{i_2} - R_j), \tag{D.15}$$

$$\hat{\gamma}_{TRR} = \frac{2}{n_T(n_R - 1)n_R} \sum_{i=1}^{n_T} \sum_{j_1=1}^{n_R-1} \sum_{j_2=j_1+1}^{n_R} I^+(T_i - R_{j_1}) \cdot I^+(T_i - R_{j_2}). \tag{D.16}$$

In the case that larger values of $\hat{\gamma}$ are considered inferior, then we reject the null hypothesis of inferiority if

$$\frac{\left(\frac{1}{2} + e\right) - \hat{\gamma}}{\hat{\sigma}_{\hat{\gamma}}} > z_{1-\alpha}. \tag{D.17}$$

If instead smaller values are inferior, the null is rejected if

$$\frac{\hat{\gamma} - \left(\frac{1}{2} - e\right)}{\hat{\sigma}_{\hat{\gamma}}} > z_{1-\alpha}. \tag{D.18}$$

In both cases, $z_{1-\alpha}$ is the $1-\alpha$ quantile of the standard normal distribution.

Appendix E

Discrepancies with Iliev et al. 2006

For the CRTCP codes whose data were re-analysed in Chapter 5, Section 5, comparison to Iliev et al. (2006), henceforth IL06, shows that results are generally identical. For completeness, several discrepancies and inconsistencies are listed and explained here.

First, IFT histograms of the ionization fraction for test II, in Section 5.2, differ. The results of IL06 are consistent with histograms of the neutral fraction, but with all $x_{\mathrm{HI}} = 1$ cells first removed.

For test III, one minor inconsistency is that IL06 report radiation to be incident from the $y = 0$ face, with the clump located at (5, 3.3, 3.3)kpc. However, their results make it clear that either radiation is incident from $x = 0$, or that the clump is located at (3.3, 5, 3.3)kpc. Here, the former was assumed. In any case, the choice should not have any appreciable impact on results after the low-density region becomes ionized, which occurs well before the first output.

Finally, test IV weighted ionization fractions differ. In particular, IL06 do not observe a crossing of the mass- and volume-weighted averages. They also specify output times in units of an unspecified t_{rec}, with values of approximately 0.1, 0.25, 0.8, 1.6 and 4.1. Regardless of units, the delta between consecutive values is not consistent with the test IV snapshots at 0.05, 0.1, 0.2, 0.3 and 0.4 Myr.

It so happens that, while downloading all available data from the comparison project website,[1] Eric Tittley noticed some of the analysis code used is also available. This shows first of all that the output times of the test I and II results have erroneously been

[1] `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project`, accessed 2017-07-18.

used for test IV.[2] It also shows that the densities required for mass-weighted averages were interpreted incorrectly.

The provided density file contains 130 Fortran records. The first contains the redshift as a single-precision value. The second is a single-precision dummy value. The remaining 128 are sequences of 128^2 single-precision values, each representing a slice of the density grid at constant z co-ordinate. The x co-ordinate varies fastest for these values. This is specified as much on the comparison project website,[3] which provides the below Fortran code for reading in the test IV density file. I have also verified that this is a correct interpretation, with high-density areas corresponding to the provided list of source locations.

```
do k=1,ngrid
   read(1) ((rho(i,j,k),i=1,ngrid),
                          j=1,ngrid)
end do
```

However, examination of the analysis code shows that the below is used.[4]

```
do i=1,n
   read(1) ((rho(i,j,k),k=1,n),j=1,n)
end do
```

Clearly, the x and z co-ordinates have been permuted, and this is not later corrected or accounted for. Thus the masses used to weight the ionization fractions in IL06 for test IV are essentially uncorrelated with the masses of the corresponding cells. Such a permutation does indeed reproduce Figure 35 of IL06, axis labels notwithstanding.

[2] See `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project/RT_workshop_data/T4_results/fracs.sm`, accessed 2017-07-18.

[3] See `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project/files/RT_tests.pdf`, accessed 2017-07-18.

[4] See for example `https://astronomy.sussex.ac.uk/~iti20/RT_comparison_project/RT_workshop_data/T4_results/Mellema/RT_global_T4.f`, accessed 2017-07-18. However, all codes use an identical file, and 'Mellema' may be replaced with 'Crash', 'Razoumov' or 'Ritzerveld'.

Bibliography

Abel, T., Norman, M. L., and Madau, P. (1999). Photon-conserving radiative transfer around point sources in multidimensional numerical cosmology. *Astrophys. J.*, 523(1):66–71.

Abel, T. and Wandelt, B. D. (2002). Adaptive ray tracing for radiative transfer around point sources. *Mon. Not. R. Astron. Soc.*, 330(3):53–56.

Agertz, O., Moore, B., Stadel, J., Potter, D., Miniati, F., Read, J., Mayer, L., Gawryszczak, A., Kravtsov, A., Nordlund, Å., Pearce, F., Quilis, V., Rudd, D., Springel, V., Stone, J., Tasker, E., Teyssier, R., Wadsley, J., Walder, R., Vej, J. M., and Ø, D.-K. (2007). Fundamental differences between SPH and grid methods. *Mon. Not. R. Astron. Soc.*, 380(3):963–978.

Aila, T., Karras, T., and Laine, S. (2013). On quality metrics of bounding volume hierarchies. *Proc. 5th High-Performance Graph. Conf. - HPG '13*, page 101.

Aila, T. and Laine, S. (2009). Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. Conf. High Perform. Graph. 2009*, HPG '09, pages 145–149, New York, NY, USA. ACM.

Aila, T., Laine, S., and Karras, T. (2012). Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation.

Aldrovandi, S. M. V. and Péquignot, D. (1973). Radiative and Dielectronic Recombination Coefficients for Complex Ions. *Astron. Astrophys.*, 25:137–140.

Altay, G., Croft, R., and Pelupessy, I. (2008). SPHRAY: a smoothed particle hydrodynamics ray tracer for radiative transfer. *Mon. Not. R. Astron. Soc.*, 386(4):1931–1946.

Altay, G. and Theuns, T. (2013). URCHIN: a reverse ray tracer for astrophysical applications. *Mon. Not. R. Astron. Soc.*, 434(1):748–764.

Altay, G. and Wise, J. H. (2015). Rabacus: A Python package for analytic cosmological radiative transfer calculations. *Astron. Comput.*, 10:73–87.

Alvarez, M. A., Bromm, V., and Shapiro, P. R. (2006). The H II Region of the First Star. *Astrophys. J.*, 639(2):621.

Amanatides, J. and Woo, A. (1987). A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87*, pages 3–10.

Anninos, P., Zhang, Y., Abel, T., and Norman, M. L. (1997). Cosmological hydrodynamics with multi-species chemistry and nonequilibrium ionization and cooling. *New Astron.*, 2(3):209–224.

Apetrei, C. (2014). Fast and Simple Agglomerative LBVH Construction. In Borgo, R. and Tang, W., editors, *Comput. Graph. Vis. Comput.* The Eurographics Association.

Appel, A. (1968). Some Techniques for Shading Machine Renderings of Solids. In *Proc. April 30–May 2, 1968, Spring Jt. Comput. Conf.*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA. ACM.

Aubert, D. and Teyssier, R. (2008). A radiative transfer scheme for cosmological reionization based on a local Eddington tensor. *Mon. Not. R. Astron. Soc.*, 387(1):295–307.

Aubert, D. and Teyssier, R. (2010). Reionization Simulations Powered by Graphics Processing Units. I. On the Structure of the Ultraviolet Radiation Field. *Astrophys. J.*, 724(1):244.

Bagla, J. S. (2005). Cosmological N-body simulation: Techniques, scope and status. *Curr. Sci.*, 88(7):1088–1100.

Bard, D., Bellis, M., Allen, M. T., Yepremyan, H., and Kratochvil, J. M. (2013). Cosmological calculations on the GPU. *Astron. Comput.*, 1:17–22.

Barnes, J. and Hut, P. (1986). A hierarchical O(N Log N) force-calculation algorithm. *Nature*, 324(6096):446–449.

Barnes, J. E. (1986). An efficient N-body algorithm for a fine-grain parallel computer. In Hut, P. and McMillan, S., editors, *Use Supercomput. Stellar Dyn.*, pages 175–180, New York, NY, USA. Springer-Verlag.

Bauer, A., Springel, V., Vogelsberger, M., Genel, S., Torrey, P., Sijacki, D., Nelson, D., and Hernquist, L. (2015). Hydrogen reionization in the Illustris universe. *Mon. Not. R. Astron. Soc.*, 453(4):3594–3611.

Bédorf, J., Gaburov, E., Fujii, M. S., Nitadori, K., Ishiyama, T., and Zwart, S. P. (2014). 24.77 Pflops on a Gravitational Tree-code to Simulate the Milky Way Galaxy with 18600 GPUs. In *Proc. Int. Conf. High Perform. Comput. Networking, Storage Anal.*, SC '14, pages 54–65, Piscataway, NJ, USA. IEEE Press.

Bédorf, J., Gaburov, E., and Portegies Zwart, S. (2012). A sparse octree gravitational N-body code that runs entirely on the GPU processor. *J. Comput. Phys.*, 231(7):2825–2839.

Bentley, J. L. (1975). Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517.

Berger, M. J. and Colella, P. (1989). Local Adaptive Mesh Refinement for Shock Hydrodynamics. *J. Comput. Phys.*, 82(1):64–84.

Berger, R. L. and Hsu, J. C. (1996). Bioequivalence trials, intersection-union tests and equivalence confidence sets. *Stat. Sci.*, 11(4):283–319.

Bertschinger, E. (1998). Simulations of structure formation in the Universe. *Annu. Rev. Astron. Astrophys.*, 36:599–654.

Bikker, J. (2007). Real-time Ray Tracing through the Eyes of a Game Developer. In *2007 IEEE Symp. Interact. Ray Tracing*, pages 1–10. IEEE.

Bodenheimer, P., Yorke, H. W., Rozyczka, M., and Tohline, J. E. (1990). The formation phase of the solar nebula. *Astrophys. J.*, 355:651–660.

Bolton, J., Meiksin, A., and White, M. (2004). Radiative transfer through the intergalactic medium. *Mon. Not. R. Astron. Soc.*, 348(3):L43–L48.

Cantalupo, S. and Porciani, C. (2011). RADAMESH: cosmological radiative transfer for Adaptive Mesh Refinement simulations. *Mon. Not. R. Astron. Soc.*, 411(3):1678–1694.

Carr, N. A., Hall, J. D., and Hart, J. C. (2002). The Ray Engine. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graph. Hardw.*, HWWS '02, pages 37–46, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Carr, N. A., Hoberock, J., Crane, K., and Hart, J. C. (2006). Fast GPU Ray Tracing of Dynamic Meshes Using Geometry Images. In *Proc. Graph. Interface 2006*, GI '06, pages 203–209, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

Cavuoti, S., Garofalo, M., Brescia, M., Paolillo, M., Pescape', A., Longo, G., and Ventre, G. (2014). Astrophysical data mining with GPU. A case study: Genetic classification of globular clusters. *New Astron.*, 26:12–22.

Cen, R. (1992). A hydrodynamic approach to cosmology: Methodology. *Astrophys. J. Suppl. Ser.*, 78:341–364.

Chen, P., Norman, M. L., Xu, H., and Wise, J. H. (2017). Fully Coupled Simulation of Cosmic Reionization. III. Stochastic Early Reionization by the Smallest Galaxies. *pre-print*.

Ciardi, B., Bolton, J. S., Maselli, A., and Graziani, L. (2012). The effect of intergalactic helium on hydrogen reionization: implications for the sources of ionizing photons at $z > 6$. *Mon. Not. R. Astron. Soc.*, 423(1):558–574.

Ciardi, B., Ferrara, A., Marri, S., and Raimondo, G. (2001). Cosmological reionization around the first stars: Monte Carlo radiative transfer. *Mon. Not. R. Astron. Soc.*, 324(2):381–388.

Cleary, J. G. and Wyvill, G. (1988). Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Vis. Comput.*, 4(2):65–83.

Commerçon, B., Teyssier, R., Audit, E., Hennebelle, P., and Chabrier, G. (2011). Radiation hydrodynamics with adaptive mesh refinement and application to prestellar core collapse. *Astron. Astrophys.*, 529:A35.

Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed Ray Tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145.

Cunningham, A. J., Klein, R. I., Krumholz, M. R., and McKee, C. F. (2011). Radiation-Hydrodynamic Simulations of Massive Star Formation with Protostellar Outflows. *Astrophys. J.*, 740(2):107.

D. Stamatellos and A. P. Whitworth (2005). Monte Carlo radiative transfer in SPH density fields. *Astron. Astrophys.*, 439(1):153–158.

Dehnen, W. and Read, J. I. (2011). N-body simulations of gravitational dynamics. *Eur. Phys. J. Plus*, 126(5):55.

Diggle, P. J., Fisher, N. I., and Lee, A. J. (1985). A comparison of tests of uniformity for spherical data. *Aust. J. Stat.*, 27(1):53–59.

Dixon, P. M. (2002). Ripley's K function. In El-Shaarawi, A. H. and Piegorsch, W. W., editors, *Encycl. Environmetrics*, volume 3, pages 1796–1803. John Wiley & Sons Ltd., Chichester.

Djeu, P., Hunt, W., Wang, R., Elhassan, I., Stoll, G., and Mark, W. R. (2011). Razor: An Architecture for Dynamic Multiresolution Ray Tracing. *ACM Trans. Graph.*, 30(5):115:1—-115:26.

Eisemann, M., Magnor, M., Grosch, T., and Müller, S. (2007). Fast Ray / Axis-Aligned Bounding Box Overlap Tests using Ray Slopes. *J. Graph. GPU, Game Tools*, 12(4):35–46.

Feng, Y., Croft, R. A. C., Matteo, T. D., and Khandai, N. (2013). Growth and anisotropy of ionization fronts near high-redshift quasars in the MassiveBlack simulation. *Mon. Not. R. Astron. Soc.*, 429(2):1554–1563.

Ferland, G. J., Korista, K. T., Verner, D. A., Ferguson, J. W., Kingdon, J. B., and Verner, E. M. (1998). Invited Review CLOUDY 90: Numerical Simulation of Plasmas and Their Spectra. *Publ. Astron. Soc. Pacific*, 110(749):761–778.

Finlator, K., Özel, F., and Davé, R. (2009). A new moment method for continuum radiative transfer in cosmological re-ionization. *Mon. Not. R. Astron. Soc.*, 393(4):1090–1106.

Fisher, N. I., Lewis, T., and Embleton, B. J. J. (1987). *Statistical Analysis of Spherical Data:*. Cambridge University Press, Cambridge.

Foley, T. and Sugerman, J. (2005). KD-tree Acceleration Structures for a GPU Raytracer. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graph. Hardw.*, HWWS '05, pages 15–22, New York, NY, USA. ACM.

Forgan, D. and Rice, K. (2010). Native synthetic imaging of smoothed particle hydrodynamics density fields using gridless Monte Carlo radiative transfer. *Mon. Not. R. Astron. Soc.*, 406(4):2549–2558.

Friedrich, M. M., Mellema, G., Iliev, I. T., and Shapiro, P. R. (2012). Radiative transfer of energetic photons: X-rays and helium ionization in C2-Ray. *Mon. Not. R. Astron. Soc.*, 421(3):2232–2250.

Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., Macneice, P., Rosner, R., Truran, J. W., and Tufo, H. (2000). FLASH: An Adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophys. J. Suppl. Ser.*, 131:273–334.

Furlanetto, S., Lidz, A., Loeb, A., McQuinn, M., Pritchard, J., Alvarez, M., Backer, D., Bowman, J., Burns, J., Carilli, C., Cen, R., Cooray, A., Gnedin, N., Greenhill, L., Haiman, Z., Hewitt, J., Hirata, C., Lazio, J., Mesinger, A., Madau, P., Morales, M., Oh, S., Peterson, J., Pihlström, Y., Shapiro, P., Tegmark, M., Trac, H., Zahn, O.,

and Zaldarriaga, M. (2009). Astrophysics from the Highly-Redshifted 21 cm Line. In *astro2010 Astron. Astrophys. Decad. Surv.*, volume 2010 of *Astronomy*.

Furlanetto, S. R. and Stoever, S. J. (2010). Secondary ionization and heating by fast electrons. *Mon. Not. R. Astron. Soc.*, 1878(4):1869–1878.

Fussell, D. S. and Subramanian, K. R. (1988). Fast Ray Tracing Using K-d Trees. Technical Report TR-88-07, Department of Computer Science, University of Texas at Austin, Austin, TX, USA.

Garanzha, K. and Loop, C. (2010). Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Comput. Graph. Forum*, 29(2):289–298.

Garanzha, K., Pantaleoni, J., and McAllister, D. (2011). Simpler and faster HLBVH with work queues. *Proc. ACM SIGGRAPH Symp. High Perform. Graph. - HPG '11*, page 59.

Gnedin, N. Y. and Abel, T. (2001). Multi-dimensional cosmological radiative transfer with a Variable Eddington Tensor formalism. *New Astron.*, 6(7):437–455.

Gnedin, N. Y. and Ostriker, J. P. (1997). Reionization of the universe and the early production of metals. *Astrophys. J.*, 486(2):581–598.

Goldsmith, J. and Salmon, J. (1987). Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20.

González, M., Audit, E., and Huynh, P. (2007). HERACLES: a three-dimensional radiation hydrodynamics code. *Astron. Astrophys.*, 464(2):429–435.

Górski, K. M., Hivon, E., Banday, A. J., Wandelt, B. D., and Hansen, F. K. (2005). HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. *Astrophys. J.*, 622:759–771.

Gould, R. J. and Thakur, R. K. (1970). Atomic processes in a low-density hydrogen-helium plasma. *Ann. Phys. (N. Y).*, 61(2):351–386.

Graziani, L., Maselli, A., and Ciardi, B. (2013). CRASH3: cosmological radiative transfer through metals. *Mon. Not. R. Astron. Soc.*, 431(1):722–740.

Greif, T. H. (2014). Multifrequency radiation hydrodynamics simulations of H2 line emission in primordial, star-forming clouds. *Mon. Not. R. Astron. Soc.*, 444(2):1566–1583.

Grimm, S. L. and Stadel, J. G. (2014). The GENGA code: gravitational encounters in N-body simulations with GPU acceleration. *Astrophys. J.*, 796(1):23.

Günther, J. (2014). *Ray tracing of dynamic scenes*. PhD thesis, Universität des Saarlandes, Saarbrücken.

Gunther, J., Popov, S., Seidel, H.-P., and Slusallek, P. (2007). Realtime Ray Tracing on GPU with BVH-based Packet Traversal. *2007 IEEE Symp. Interact. Ray Tracing*, pages 113–118.

Hassan, a. H., Fluke, C. J., Barnes, D. G., and Kilborn, V. a. (2013). Tera-scale astronomical data analysis and visualization. *Mon. Not. R. Astron. Soc.*, 429(3):2442–2455.

Havran, V. (2001). *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Czech Technical University in Prague.

Havran, V. and Bittner, J. (2002). On Improving KD-Trees for Ray Shooting. In *Proc. WSCG 2002 Conf.*, pages 209–217.

Heggie, D. and Hut, P. (2003). *The Gravitational Million-Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics*. Cambridge University Press, Cambridge.

Heymann, F. and Siebenmorgen, R. (2012). GPU-based Monte Carlo Dust Radiative Transfer Scheme Applied to Active Galactic Nuclei. *Astrophys. J.*, 751(1):27.

Hilbert, S., Hartlap, J., White, S. D. M., and Schneider, P. (2009). Ray-tracing through the Millennium Simulation: Born corrections and lens-lens coupling in cosmic shear and galaxy-galaxy lensing. *Astron. Astrophys.*, 499(1):31–43.

Hillis, W. D. and Steele Jr., G. L. (1986). Data Parallel Algorithms. *Commun. ACM*, 29(12):1170–1183.

Hopkins, P. F. (2015). A new class of accurate, mesh-free hydrodynamic simulation methods. *Mon. Not. R. Astron. Soc.*, 450(1):53–110.

Hubber, D. A., Falle, S. A. E. G., and Goodwin, S. P. (2013). Convergence of AMR and SPH simulations – I. Hydrodynamical resolution and convergence tests. *Mon. Not. R. Astron. Soc.*, 432(1):711–727.

Hui, L. and Gnedin, N. Y. (1997). Equation of state of the photoionized intergalactic medium. *Mon. Not. R. Astron. Soc.*, 292(1):27–42.

Hummer, D. G. and Storey, P. J. (1998). Recombination of helium-like ions - I. Photoionization cross-sections and total recombination and cooling coefficients for atomic helium. *Mon. Not. R. Astron. Soc.*, 297(4):1073–1078.

Hutter, A., Dayal, P., Müller, V., and Trott, C. M. (2017). Exploring 21 cm-Lyα Emitter Synergies for SKA. *Astrophys. J.*, 836(2):176.

Iliev, I. T., Ciardi, B., Alvarez, M. A., Maselli, A., Ferrara, A., Gnedin, N. Y., Mellema, G., Nakamoto, T., Norman, M. L., Razoumov, A. O., Rijkhorst, E.-J., Ritzerveld, J., Shapiro, P. R., Susa, H., Umemura, M., and Whalen, D. J. (2006). Cosmological radiative transfer codes comparison project – I. The static density field tests. *Mon. Not. R. Astron. Soc.*, 371(3):1057–1086.

Ize, T. (2009). *Efficient Acceleration Structures for Ray Tracing Static and Dynamic Scenes*. PhD thesis, Salt Lake City, UT, USA.

Kajiya, J. T. (1986). The Rendering Equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150.

Kalojanov, J. and Slusallek, P. (2009). A Parallel Algorithm for Construction of Uniform Grids. In *Proc. Conf. High Perform. Graph. 2009*, HPG '09, pages 23–28, New York, NY, USA. ACM.

Karras, T. (2012). Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proc. Fourth ACM SIGGRAPH / Eurographics Conf. High-Performance Graph.*, EGGH-HPG'12, pages 33–37, Aire-la-Ville, Switzerland. Eurographics Association.

Karras, T. and Aila, T. (2013). Fast parallel construction of high-quality bounding volume hierarchies. *Proc. 5th High-Performance Graph. Conf. - HPG '13*, page 89.

Kessel-Deynet, O. and Burkert, A. (2000). Ionizing radiation in smoothed particle hydrodynamics. *Mon. Not. R. Astron. Soc.*, 315(4):713–721.

Killedar, M., Lasky, P. D., Lewis, G. F., and Fluke, C. J. (2012). Gravitational lensing with three-dimensional ray tracing. *Mon. Not. R. Astron. Soc.*, 420(1):155–169.

Kimm, T., Katz, H., Haehnelt, M., Rosdahl, J., Devriendt, J., and Slyz, A. (2017). Feedback-regulated star formation and escape of LyC photons from mini-haloes during reionization. *Mon. Not. R. Astron. Soc.*, 466(4):4826–4846.

Klosowski, J., Held, M., Mitchell, J., Sowizral, H., and Zikan, K. (1998). Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Trans. Vis. Comput. Graph.*, 4(1):21–36.

Komatsu, E., Smith, K. M., Dunkley, J., Bennett, C. L., Gold, B., Hinshaw, G., Jarosik, N., Larson, D., Nolta, M. R., Page, L., Spergel, D. N., Halpern, M., Hill, R. S., Kogut, A., Limon, M., Meyer, S. S., Odegard, N., Tucker, G. S., Weiland, J. L., Wollack, E., and Wright, E. L. (2011). Seven-year Wilkinson Microwave Anisotropy Prope (WMAP) observations: cosmological interpretation. *Astrophys. J. Suppl. Ser.*, 192(2):18.

Kuiper, R., Klahr, H., Dullemond, C., Kley, W., and Henning, T. (2010). Fast and accurate frequency-dependent radiation transport for hydrodynamics simulations in massive star formation. *Astron. Astrophys.*, 81:16.

Kuiper, R. and Klessen, R. S. (2013). The reliability of approximate radiation transport methods for irradiated disk studies. *Astron. Astrophys.*, 555:A7.

Kunasz, P. and Auer, L. H. (1988). Short characteristic integration of radiative transfer problems: Formal solution in two-dimensional slabs. *J. Quant. Spectrosc. Radiat. Transf.*, 39(1):67–79.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast BVH Construction on GPUs. *Comput. Graph. Forum*, 28(2):375–384.

Lauterbach, C., Yoon, S.-E., Tuft, D., and Manocha, D. (2006). RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *IEEE Symp. Interact. Ray Tracing 2006*, pages 39–46.

L'Ecuyer, P. (1999). Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators. *Oper. Res.*, 47(1):159–164.

L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. (2002). An Object-Oriented Random-Number Package with Many Long Streams and Substreams. *Oper. Res.*, 50(6):1073–1075.

Levermore, C. D. (1984). Relating Eddington factors to flux limiters. *J. Quant. Spectrosc. Radiat. Transf.*, 31(2):149–160.

Livermore, R. C., Finkelstein, S. L., and Lotz, J. M. (2017). Directly Observing the Galaxies Likely Responsible for Reionization. *Astrophys. J.*, 835(2):113.

MacDonald, D. J. and Booth, K. S. (1990). Heuristics for Ray Tracing Using Space Subdivision. *Vis. Comput.*, 6(3):153–166.

Mackey, J. (2012). Accuracy and efficiency of raytracing photoionisation algorithms. *Astron. Astrophys.*, 539:A147.

Mahovsky, J. and Wyvill, B. (2004). Fast ray-axis aligned bounding box overlap tests with plucker coordinates. *J. Graph. Tools*, 9(1):35–46.

Marsaglia, G. (2003). Xorshift RNGs. *J. Stat. Softw.*, 8(1):1–6.

Maselli, A., Ciardi, B., and Kanekar, A. (2009). Crash2: Coloured Packets and Other Updates. *Mon. Not. R. Astron. Soc.*, 393(1):171–178.

Maselli, A., Ferrara, A., and Ciardi, B. (2003). Crash: a Radiative Transfer Scheme. *Mon. Not. R. Astron. Soc.*, 345(2):379–394.

McKinney, J. C., Tchekhovskoy, A., Sadowski, A., and Narayan, R. (2014). Three-dimensional general relativistic radiation magnetohydrodynamical simulation of super-Eddington accretion, using a new code HARMRAD with M1 closure. *Mon. Not. R. Astron. Soc.*, 441(4):3177–3208.

Mellema, G., Iliev, I. T., Alvarez, M. A., and Shapiro, P. R. (2006). C2-ray: A new method for photon-conserving transport of ionizing radiation. *New Astron.*, 11(5):374–395.

Möller, T. and Trumbore, B. (2005). Fast, Minimum Storage Ray/Triangle Intersection. *ACM SIGGRAPH 2005 Courses*, (1):1–7.

Monaghan, J. (1992). Smoothed Particle Hydrodynamics. *Annu. Rev. Astron. Astrophys.*, 30(1):543–574.

Monaghan, J. (2005). Smoothed particle hydrodynamics. *Reports Prog. Phys.*, 68(8):1703–1759.

Müller, G. and Fellner, D. (1999). Hybrid scene structuring with application to ray tracing. In *Proc. Int. Conf. Vis. Comput.*, pages 19–26.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53.

Norman, M., Paschos, P., and Abel, T. (1998). Simulating inhomogeneous reionization. *Mem. della Soc. Astron. Ital.*, 69:455.

Norman, M. L., Reynolds, D. R., So, G. C., Harkness, R. P., and Wise, J. H. (2015). Fully coupled simulation of cosmic reionization. I. numerical methods and tests. *Astrophys. J. Suppl. Ser.*, 216(1):16.

Ocvirk, P., Gillet, N., Shapiro, P. R., Aubert, D., Iliev, I. T., Teyssier, R., Yepes, G., Choi, J.-H., Sullivan, D., Knebe, A., Gottlöber, S., D'Aloisio, A., Park, H., Hoffman, Y., and Stranex, T. (2016). Cosmic Dawn (CoDa): the first radiation-hydrodynamics simulation of reionization and galaxy formation in the Local Universe. *Mon. Not. R. Astron. Soc.*, 463(2):1462–1485.

Okumura, A., Hayashida, M., Katagiri, H., Saito, T., and Vassiliev, V. (2011). Development of Non-sequential Ray-tracing Software for Cosmic-ray Telescopes. In *32nd Int. Cosm. Ray Conf.*, volume 9, page 211.

O'Shea, B. W., Wise, J. H., Xu, H., and Norman, M. L. (2015). Probing the Ultraviolet Luminosity Function of the Earliest Galaxies With the Renaissance Simulations. *Astrophys. J.*, 807(1):L12.

Osterbrock, D. (1989). *Astrophysics of Gaseous Nebulae and Active Galactic Nuclei*. University Science Books, Mill Valley, California.

Oxley, S. and Woolfson, M. M. (2003). Smoothed particle hydrodynamics with radiation transfer. *Mon. Not. R. Astron. Soc.*, 343(3):900–912.

Paardekooper, J.-P. (2010). *And there was light: Voronoi-Delaunay radiative transfer and cosmic reionisation*. PhD thesis, Leiden University.

Paardekooper, J.-P., Khochfar, S., and Dalla, C. V. (2012). The First Billion Years project: proto-galaxies reionizing the Universe. *Mon. Not. R. Astron. Soc. Lett.*, 429(1):L94–L98.

Paardekooper, J. P., Khochfar, S., and Vecchia, C. D. (2015). The first billion years project: The escape fraction of ionizing photons in the epoch of reionization. *Mon. Not. R. Astron. Soc.*, 451(3):2544–2563.

Paardekooper, J.-P., Kruip, C. j. H., and Icke, V. (2010). SimpleX2: radiative transfer on an unstructured, dynamic grid. *Astron. Astrophys.*, 515:A79.

Pantaleoni, J. and Luebke, D. (2010). HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *Proc. Conf. High Perform. Graph.*, HPG '10, pages 87–95, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.*

Parsons, A. R., Liu, A., Aguirre, J. E., Ali, Z. S., Bradley, R. F., Carilli, C. L., Deboer, D. R., Dexter, M. R., Gugliucci, N. E., Jacobs, D. C., Klima, P., MacMahon, D. H. E., Manley, J. R., Moore, D. F., Pober, J. C., Stefan, I. I., and Walbrugh, W. P. (2014). New Limits on 21 cm Epoch of Reionization from PAPER-32 Consistent with an X-Ray Heated Intergalactic Medium at $z = 7.7$. *Astrophys. J.*, 788(2010):106.

Partl, A. M., Maselli, A., Ciardi, B., Ferrara, A., and Müller, V. (2011). Enabling parallel computing in crash. *Mon. Not. R. Astron. Soc.*, 414(1):428–444.

Pawlik, A. H. and Schaye, J. (2008). TRAPHIC - Radiative transfer for smoothed particle hydrodynamics simulations. *Mon. Not. R. Astron. Soc.*, 389(2):651–677.

Pawlik, A. H. and Schaye, J. (2011). Multifrequency, thermally coupled radiative transfer with TRAPHIC: method and tests. *Mon. Not. R. Astron. Soc.*, 412(3):1943–1964.

Peebles, P. J. E. (1968). Recombination of the primeval plasma and light -inos. *Astrophys. J.*, 153:1.

Peters, T. (2014). The physics of volume rendering. *Eur. J. Phys.*, 35(6):65028.

Petkova, M. and Springel, V. (2009). An implementation of radiative transfer in the cosmological simulation code gadget. *Mon. Not. R. Astron. Soc.*, 396(3):1383–1403.

Pharr, M. and Humphreys, G. (2010). *Physically based rendering: from theory to implementation*. Morgan Kaufmann, 2nd edition.

Planck Collaboration (2014). Planck 2013 results. XVI. Cosmological parameters. *Astron. Astrophys.*, 571:A16.

Planck Collaboration (2016). Planck 2015 results. XIII. Cosmological parameters. *Astron. Astrophys.*, 594:A13.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray Tracing on Programmable Graphics Hardware. *ACM Trans. Graph.*, 21(3):703–712.

Razoumov, A. O. and Cardall, C. Y. (2005). Fully threaded transport engine: New method for multi-scale radiative transfer. *Mon. Not. R. Astron. Soc.*, 362(4):1413–1417.

Razoumov, A. O. and Sommer-Larsen, J. (2006). Escape of Ionizing Radiation from Star-forming Regions in Young Galaxies. *Astrophys. J. Lett.*, 651(2):L89.

Reinhard, E., Smits, B., and Hansen, C. (2000). Dynamic Acceleration Structures for Interactive Ray Tracing. In Péroche, B. and Rushmeier, H., editors, *Render. Tech. 2000*, Eurographics, pages 299–306. Springer Vienna.

Reshetov, A., Soupikov, A., and Hurley, J. (2005). Multi-level Ray Tracing Algorithm. *ACM Trans. Graph.*, 24(3):1176–1185.

Rijkhorst, E., Plewa, T., Dubey, A., and Mellema, G. (2006). Hybrid characteristics: 3D radiative transfer for parallel adaptive mesh refinement hydrodynamics. *Astron. Astrophys.*, 452:907–920.

Ritzerveld, J. (2005). The diffuse nature of Strömgren spheres. *Astron. Astrophys.*, 439:L23–L26.

Ritzerveld, J. and Icke, V. (2006). Transport on adaptive random lattices. *Phys. Rev. E*, 74(2):026704.

Rivi, M., Gheller, C., Dykes, T., Krokos, M., and Dolag, K. (2014). GPU accelerated particle visualization with Splotch. *Astron. Comput.*, 5:9–18.

Robertson, B. E., Ellis, R. S., Dunlop, J. S., McLure, R. J., and Stark, D. P. (2010). Early star-forming galaxies and the reionization of the Universe. *Nature*, 468(7320):49–55.

Robertson, B. E., Ellis, R. S., Furlanetto, S. R., and Dunlop, J. S. (2015). Cosmic reionization and early star-forming galaxies: a joint analysis of new constraints from Planck and the Hubble Space Telescope. *Astrophys. J.*, 802(2):L19.

Robertson, B. E., Furlanetto, S. R., Schneider, E., Charlot, S., Ellis, R. S., Stark, D. P., McLure, R. J., Dunlop, J. S., Koekemoer, A., Schenker, M. A., Ouchi, M., Ono, Y., Curtis-Lake, E., Rogers, A. B., Bowler, R. A. A., and Cirasuolo, M. (2013). New Constraints on Cosmic Reionization From the 2012 Hubble Ultra Deep Field Campaign. *Astrophys. J.*, 768(1):71.

Robeson, S. M., Li, A., and Huang, C. (2014). Point-pattern analysis on the sphere. *Spat. Stat.*, 10:76–86.

Rosdahl, J., Blaizot, J., Aubert, D., Stranex, T., and Teyssier, R. (2013). RAMSES-RT: radiation hydrodynamics in the cosmological context. *Mon. Not. R. Astron. Soc.*, 436(3):2188–2231.

Roth, N. and Kasen, D. (2015). Monte Carlo Radiation Hydrodynamics with Implicit Methods. *Astrophys. J. Suppl. Ser.*, 217(1):9.

Rybicki, G. and Lightman, A. (1979). *Radiative processes in astrophysics*. Wiley-Interscience, New York.

Saftly, W., Baes, M., and Camps, P. (2014). Astrophysics Hierarchical octree and k-d tree grids for 3D radiative transfer simulations. *Astron. Astrophys.*, 561:A77.

Saftly, W., Camps, P., Baes, M., Gordon, K. D., Vandewoude, S., Rahimi, A., and Stalevski, M. (2013). Using hierarchical octrees in Monte Carlo radiative transfer simulations. *Astron. Astrophys.*, 554:A10.

Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). Parallel Random Numbers: As Easy As 1, 2, 3. In *Proc. 2011 Int. Conf. High Perform. Comput. Networking, Storage Anal.*, SC '11, pages 16:1—-16:12, New York, NY, USA. ACM.

Schive, H.-Y., Tsai, Y.-C., and Chiueh, T. (2010). Gamer: a Graphic Processing Unit Accelerated Adaptive Mesh Refinement Code for Astrophysics. *Astrophys. J. Suppl. Ser.*, 186(2):457–484.

Scholz, T. T. and Walters, H. R. J. (1991). Collisional rates and cooling within atomic hydrogen plasmas. *Astrophys. J. Lett. v.489*, 380:302.

Schuirmann, D. J. (1987). A comparison of the Two One-Sided Tests Procedure and the Power Approach for assessing the equivalence of average bioavailability. *J. Pharmacokinet. Biopharm.*, 15(6):657–680.

Seaton, M. (1959). Radiative recombination of hydrogenic ions. *Mon. Not. R. Astron. Soc.*, 119:81.

Shao, M., Nemati, B., Zhai, C., Turyshev, S. G., Sandhu, J., Hallinan, G., and Harding, L. K. (2014). Finding Very Small Near-Earth Asteroids Using Synthetic Tracking. *Astrophys. J.*, 782(1):1.

Shapiro, P. R., Iliev, I. T., Mellema, G., Ahn, K., Mao, Y., Friedrich, M., Datta, K., Park, H., Komatsu, E., Fernandez, E., Koda, J., Bovill, M., and Pen, U.-L. (2012). Simulating cosmic reionization and the radiation backgrounds from the epoch of reionization. *AIP Conf. Proc.*, 1480(1):248–260.

Shull, J. M. and van Steenberg, M. E. (1985). X-Ray secondary heating and ionization in quasar emission-line clouds. *Astrophys. J.*, 298:268–274.

Skinner, M. A. and Ostriker, E. C. (2013). A two-moment radiation hydrodynamics module in Athena using a time-explicit Godunov method. *Astrophys. J. Suppl. Ser.*, 206(2):21.

Smith, A., Safranek-Shrader, C., Bromm, V., and Milosavljević, M. (2015). The Lyman α signature of the first galaxies. *Mon. Not. R. Astron. Soc.*, 449(4):4336–4362.

Smits, B. (1998). Efficiency Issues for Ray Tracing. *J. Graph. Tools*, 3(2):1–14.

Springel, V. (2005). The cosmological simulation code gadget-2. *Mon. Not. R. Astron. Soc.*, 364(4):1105–1134.

Springel, V. (2010a). E pur si muove: Galilean-invariant cosmological hydrodynamical simulations on a moving mesh. *Mon. Not. R. Astron. Soc.*, 401(2):791–851.

Springel, V. (2010b). Smoothed Particle Hydrodynamics in Astrophysics. *Annu. Rev. Astron. Astrophys.*, 48(1):391–430.

Springel, V. and Hernquist, L. (2002). Cosmological smoothed particle hydrodynamics simulations: the entropy equation. *Mon. Not. R. Astron. Soc.*, 333(3):649–664.

Steinacker, J., Baes, M., and Gordon, K. D. (2013). Three-Dimensional Dust Radiative Transfer. *Annu. Rev. Astron. Astrophys.*, 51(1):63–104.

Stich, M., Friedrich, H., and Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. *Proc. 1st ACM Conf. High Perform. Graph. - HPG '09*, page 7.

Stone, J. E., Gohara, D., and Shi, G. (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Comput. Sci. Eng.*, 12(3):66–73.

Susa, H. (2006). Smoothed Particle Hydrodynamics Coupled with Radiation Transfer. *Publ. Astron. Soc. Japan*, 58(2):445–460.

Szirmay-Kalos, L., Havran, V., Balázs, B., and Szécsi, L. (2002). On the Efficiency of Ray-shooting Acceleration Schemes. In *Proc. 18th Spring Conf. Comput. Graph.*, SCCG '02, pages 97–106, New York, NY, USA. ACM.

Teyssier, R. (2002). Cosmological hydrodynamics with adaptive mesh refinement A new high resolution code called RAMSES. *Astron. Astrophys.*, 385:337–364.

Tittley, E. R. and Meiksin, A. (2007). Reionization scenarios and the temperature of the intergalactic medium. *Mon. Not. R. Astron. Soc.*, 380(4):1369–1386.

Tomida, K., Tomisaka, K., Matsumoto, T., Ohsuga, K., Machida, M. N., and Saigo, K. (2010). Radiation Magnetohydrodynamics Simulation of Proto-Stellar Collapse: Two-Component Molecular Outflow. *Astrophys. J. Lett.*, 714(1):L58.

Vaytet, N. M. H., Audit, E., Dubroca, B., and Delahaye, F. (2011). A numerical model for multigroup radiation hydrodynamics. *J. Quant. Spectrosc. Radiat. Transf.*, 112(8):1323–1335.

Vinkler, M., Havran, V., and Bittner, J. (2015). Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing. *Comput. Graph. Forum.*

Wald, I. (2007). On fast construction of SAH-based bounding volume hierarchies. *Proc. IEEE/EG Symp. Interact. Ray Tracing 2007*, pages 33–40.

Wald, I., Boulos, S., and Shirley, P. (2007a). Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.*, 26(1).

Wald, I. and Havran, V. (2006). On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N). In *IEEE Symp. Interact. Ray Tracing 2006*, pages 61–69.

Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. (2006). Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Trans. Graph.*, 25(3):485–493.

Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., Parker, S. G., and Shirley, P. (2007b). State of the Art in Ray Tracing Animated Scenes. In Schmalstieg, D. and Bittner, J., editors, *STAR Proc. Eurographics 2007*, pages 89–116. The Eurographics Association.

Walker, E. and Nowacki, A. S. (2011). Understanding equivalence and noninferiority testing. *J. Gen. Intern. Med.*, 26(2):192–196.

Warren, M. and Salmon, J. (1993). A parallel hashed oct-tree n-body algorithm. In *Proc. 1993 ACM/IEEE Conf. Supercomput.*, pages 12–21, New York, NY, USA. ACM.

Weiskopf, D. (2000). Four-dimensional Non-linear Ray Tracing As a Visualization Tool for Gravitational Physics. In *Proc. Conf. Vis. 2000*, VIS '00, pages 445–448, Los Alamitos, CA, USA. IEEE Computer Society Press.

Wellek, S. (1996). A new approach to equivalence assessment in standard comparative bioavailability trials by means of the Mann-Whitney statistic. *Biometrical J.*, 38:695–710.

Wellek, S. (2010). *Testing Statistical Hypotheses of Equivalence and Noninferiority*. Chapman and Hall/CRC, 2nd edition.

Whalen, D. and Norman, M. L. (2006). A Multistep Algorithm for the Radiation Hydrodynamical Transport of Cosmological Ionization Fronts and Ionized Flows. *Astrophys. J. Suppl. Ser.*, 162:281.

Whitted, T. (1980). An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349.

Williams, A., Barrus, S., Morley, R. K., and Shirley, P. (2005). An efficient and robust ray-box intersection algorithm. In *ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA. ACM.

Wise, J. H. and Abel, T. (2011). Enzo+Moray: Radiation Hydrodynamics Adaptive Mesh Refinement Simulations With Adaptive Ray Tracing. *Mon. Not. R. Astron. Soc.*, 414(4):3458–3491.

Ylitie, H., Karras, T., and Laine, S. (2017). Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs. In *Proc. High Perform. Graph.*, HPG '17, pages 4:1–4:13, New York, NY, USA. ACM.

Yoon, S.-E., Curtis, S., and Manocha, D. (2007). Ray Tracing Dynamic Scenes Using Selective Restructuring. In *Proc. 18th Eurographics Conf. Render. Tech.*, EGSR'07, pages 73–84, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Zaroubi, S. (2013). *The First Galaxies: Theoretical Predictions and Observational Clues*, chapter The Epoch, pages 45–101. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zaroubi, S. and Silk, J. (2005). LOFAR as a probe of the sources of cosmological reionization. *Mon. Not. R. Astron. Soc. Lett.*, 360(1):2003–2006.